AUUG

Australian Open Systems Users Group

AUUG Summer Conference 1992

Proceedings

CONTENTS:

Softway Engineering
or
Structured UNIX Kernel Hacking

Peter Chubb

March 20, 1992

Abstract

Softway is in the process of changing the way in which we produce UNIX software. In the past, we've relied on individual expertise and commitment rather than on particular methodologies. As we've grown, and the size of the projects we undertake has grown, this is no longer adequate.

Beginning gently, project by project, we've been introducing a company-wide Software Quality program, with the eventual aim of complying with AS3563. Initial experiences have been good: we always knew we were producing good software, but could never prove it. Now we can.

1 Introduction

None of the design and software engineering methodologies that I was taught about at University seem quite to fit UNIX kernel development work.

When developing UNIX kernel enhancements, the key features are:

- A large body of existing code that mostly works.
- Few comments, and some of them misleading, in the code itself.
- No, or very little, design information about the code.
- A staff that needs specialised knowledge of a wide variety of fields that impinge upon the kernel's code.

The kinds of people that make changes to the UNIX kernel are also somewhat different from the normal software engineer (if there is such a thing). The stereotype of a bearded gentleman, rather poorly dressed, who is impatient of formality, and sees it mostly as getting in the way is not too far from reality.

As such, introducing Quality techniques for UNIX kernel development poses a few special problems. In the remainder of this document I give a (necessarily brief) summary of some of the ways we at Softway are trying to improve the quality and the perceived quality of our software.

2 Traditional Softway Software Development

Software engineering is the process by which quality software is developed to a budget. As such, it involves being able to

- extract from customers a good idea of what they really want.
- estimate the time and effort involved in producing a given piece of software.
- produce bug-free (or almost bug-free) software with the bare minimum of resources.

At Softway this has traditionally been done by relying on individual expertise. Softway has always employed some very highly skilled individuals; people who are able to look at an incomplete requirements specification and say, 'Oh, that'll take fifteen person-months', and others who can code directly from a loose requirements specification and produce code that is readable, well-commented, and close to bug free.

As we've grown, we have been tackling bigger projects — ones that are closer to ten person-years than one — and we've had to employ more staff to cope. This has meant that the proportion of people in the organisation who can estimate reliably, code and design on the fly, and test adequately has fallen. Moreover, as the kinds of jobs tackled have changed, the old wisdom has become inadequate.

Consequently, we've had to introduce a set of more formal techniques to ensure that the products we deliver are of the highest possible quality given the budgetary constraints we operate under.

We've now run four projects under the new regime — a total of 30 man years, or thereabouts — and so are in a good position to comment on ways of migrating to a more formal approach to software development.

3 Traditional Softway Methodologies

Traditionally, when a job arrived (usually because someone at Softway heard about a job that *might* be coming up) someone would be assigned to find out what the customer wanted — usually the same person that found out about the job in the first place.

Having found out as much as possible, that person would write up a requirements specification and project plan as a basis to quote from. If the quote was accepted by the customer, depending on the size of the project, one or more extra people would be assigned to it, and they would spend some time discussing ways and means of meeting the requirements, and would then go ahead and implement software in prototype form.

The prototype would be evaluated informally, end either thrown away or beefed up to match the requirements.

This classifies as a dynamic programming environment, according to Cem Kaner:

> A dynamic environment is characterised by a budget that is too small, a staff that is too small, a deadline that is too soon and can't be postponed for as long as it should and by a shared vision and commitment among the developers.
>
> In a dynamic environment, the quality of the product lies in the hands of the individuals designing, programming, testing and documenting it; each individual counts. Standards, specifications, committees, and change control will not assure

quality, nor are they relied on to play that role. The development team does make some agreements, but it is the commitment of the team's members to excellence, their mastery of the tools of their craft, and their ability to work together that makes the product, not the rules.

The development team has a vision of what they want to create, and a commitment to getting as close to the dream as they can. They recognise that they'll have to flesh out details by trial and error. By the time they've worked out the final details for one aspect of the product, they have a working version that one or two key people understand fully. That version is the "specification".

I've quoted these paragraphs because they show the good points about this work style — the key features for a successful outcome are:

- A small closely knit project team, so that communication is easy and informal.
- A team consisting almost entirely of very good committed programmers, so that very few bugs are introduced during coding.
- Well understood problems, such as porting UNIX or writing a device driver.

At that time, Softway was involved in research work. At one time, 30% of our staff were working on one product, which, at the time, brought in no income, and didn't really have any deadlines. It did have a team of people passionately devoted to making it the best technical product possible.

It was a UNIX kernel product. It had no fixed project plan, and therefore could never finish — it never reached a milestone, because there were none.

However, it was making progress. But the progress could not be measured.

The problem arose when we wanted to sell it to somebody...

3.1 Problems with Laissez-Fair Development

There are several major problems with this development style:

- Scope There is no way to limit the work that has to be done. If an extra feature pleases the project team, it is inserted.
- Traceability There is no way to say "*This* is to match *that* requirement," so work can be wasted in doing unnecessary things, and likewise, some requirements can be left out.
- Measurability. There is no way to tell how good the delivered code and documentation really is — so there is no way to tell if new development techniques are having any real effect. Moreover, there is no way to tell our customers how good we are.
- Quality. Everyone has off days — days when the error rate quintuples, and stupidity reigns supreme. In a laissez-faire development environment, there's no mechanism for catching major blunders like these (although errors which prevent the product from working will generally be caught, later rather than sooner).

- Testing. Without firm requirements, there's no way to do any serious integration or acceptance testing. Moreover, programmers are probably the worst people to design tests for their own code, because they are very bound up in it. They (we) find it difficult to step back and look at the entire problem. For that matter, everyone I know finds designing test strategies and test cases very tedious.

There are also a number of advantages:

- Paperwork. There isn't any. People working on a project are free to spend 100% of their time thinking, designing, programming and testing.
- Flexibility. The people working on the project can always slot some more work in to meet a new requirement, or delete one that has since become unneeded.
- Freedom For any given project the most appropriate design techniques can always be chosen. People working in this 'methodology' tend to design on-the-fly, so everyone gets to do creative work.

4 The New Methodologies

How can the advantages be kept, while eliminating the major disadvantages?

The approach taken was to take the AS3563 standard and move gradually toward development and quality assurance methodologies that would satisfy it, while attempting to retain the best points of our traditional approach.

4.1 AS3563

The standard provides a framework for developing quality software. The meaning of 'Quality' here, is "Meets Requirements".

The standard does *not* mandate particular design, analysis, project planning or quality assurance methodologies. What it *does* specify is that for any given project, the methodologies used shall be written down in the documentation for that project.

Rather than start by writing the Quality Manual that should lay down the law about all the officially approved standards for developing software at Softway, we decided to start off trying to work out what standards are appropriate for the kinds of work we do. As we're all practical types, we started off trying to use the IEEE standards for project plans (IEEE 1058) design descriptions (IEEE 1016) and requirements specifications (IEEE 830).

Use of the project plan and requirements specification standards were both reasonably successful after we worked out how to use them. The design standard was not, for reasons outlined in the next section.

4.2 Design Techniques

A design tool has inbuilt mechanisms for cross checking design against requirements. It allows one to structure one's material such that all the information necessary to think about a design at a particular level is on one or at the most two pages. It provides a framework to work in that aids creative thought. The one feature common to all good design tools is their power of expressing and manipulating complex relationships in a small space.

The IEEE standard for software design descriptions is just that — a documentation standard. Unfortunately, we didn't realise this, and tried to use it as a design tool.

A documentation standard says what sections have to be present for a complete description of a design to various groups: to the quality assurance people (we didn't have any!) to coders, and to designers of lower levels. In order to fill in the sections, *the design already has to be finished.*

We lost a lot of time because we didn't realise this, and tried to use the document standard as a design methodology — a way to think about things in order to do a design — rather than as a documentation standard.

4.2.1 Our design methodology

Most usual design methodologies in the literature are not suitable for the kinds of work we do. This work usually involves changing the UNIX kernel — a large program with well defined interfaces — and usually under the constraint that the changes be as few as possible.

To this end, the 'hooks' style of programming, and rapid-prototyping seem to be the most appropriate way to satisfy everyone. *Hooks* are function calls inserted into existing kernel code to functions that will modify the behaviour of the existing code.

The prototype, usually on an Intel 80386 machine, has the functionality (but not the performance) of the 'real thing'. Code can be protoyped quickly in the Intel kernel (using suck-it-and-see methods), and when it behaves properly, ported to the mainframes we do most of our work on.

In the excitement of getting a prototype working, quality requirements can be forgotten — anything goes if it works — but it is very important that the real thing be as close to bug free as practicable.

To this end, we introduced *reviews*.

4.3 Reviews

The most useful quality assurance method we've found is 'reviews'.

A review is an opportunity for many people, not just the author, to scrutinise a document (a document can be a piece of code, a design, a requirements analysis, a test harness — almost anything) and criticise it.

Before a review, the material to be reviewed — code, design, test-specification or whatever — is distributed to a select group of people, including:

1. the author
2. a moderator/chairman
3. some reviewers.

The reviewers are chosen based on who's available, and on who has the background to be able to review the documents effectively.

These people are given, along with the document, a checklist of things to look for. (In early attempts at reviewing, this checklist was not developed, and people mostly found spelling and grammatical errors, not real problems). They look for problems in the following main areas:

1. Conflicts with the input documents (for code, that would be the design document; for a prototype or a design, the requirements specification, etc).

2. Conflicts with the standards being used for the project (The Softway coding style guidelines, the IEEE document formats, or whatever).

3. Feasibility — whether the design, code or whatever will work properly

4. Robustness — whether the design, code or whatever will continue to work if various changes in its environment occur (like porting from a single to a multiprocessor, for example).

In addition, specific documents have their own checklists; and on a project by project basis extra items can be added to the checklists for particular project phases.

We are still learning about how to make reviews effective.

4.4 Bug Reporting

Softway personnel are notoriously bad at reporting bugs. When they find a bug, they fix it without telling anybody.

We wanted to keep track of how many bugs were found in various projects, so we [1] invented an incentive scheme.

The first person to fill in a bug report form for a new bug received a chocolate bug when the bug was confirmed by another member of the team.

This had two effects:

1. People were a lot keener to find *and report* bugs.

2. A friendly rivalry grew up around the number of bug carcasses attached to one's X-terminal, and hence the number of bugs found.

The first effect was the important one, however — with the reward only given for a completed form, some analysis could be done on the kinds of bugs being found.

We found in our first project using the technique that of the bugs we found, 45% were found in reviews, and the remainder in testing.

4.5 Initial Reactions

Initially, people didn't like all these standards, reviews and so forth. They were seen as subtracting time from the more important business of getting the next deliverable ready. However, as we get better at reviewing material, and better at using the standards, opinion is coming round to agree that reviews are valuable.

However, we have changed the way we review material. For a start, all material to be reviewed must be deskchecked by someone else first — this stops us wasting time on stupid errors. Also, code must pass `lint` with no error output before being reviewed.

[1] Lucy Chubb and Paul Brebner came up with the idea in the first place.

5 What comes next?

Our aim is to continue to improve as much as possible to produce quality kernel and systems-level UNIX software. We want AS3563 compliance, but we also wish to maintain some of the old Softway spirit. We're still working out how to do this effectively.

References

[1] *Software Quality Management System — Part 1: Requirements.* AS3563.1–1991.

[2] *Software Quality Management System — Part 2: Implementation.* AS3563.2–1991.

[3] *IEEE/ANSI Std 1058–1987 Standard for Software Project Management Plans.*

[4] *IEEE/ANSI Std 1016–1987 Software Design Descriptions.*

[5] *IEEE/ANSI Std 830–1984 Software Requirements Specifications.*

[6] Cem Kaner. *Testing Computer Software.* TAB Books Inc., 1988.

UNIX System V Release 4, Use and Administration. A User's Perspective.

Frank Crawford

Aust. Supercomputing Tech., Private Mail Bag 1, Menai 2234
(frank@atom.ansto.gov.au)

ABSTRACT

UNIX System V Release 4 has been around for some time, and versions from various vendors are now reaching the market. It is poised to become the standard version of UNIX, but what is it really like? Is it really standard? How is it different to what is currently available? What problems will you face the first time you use it?

This paper will attempt to answer those and many more questions. It will describe what it is like to use and to administer, what new features have been added, what has been removed and more importantly what has subtly changed. It will concentrate more on things that affect the ordinary user or the person trying to administer such a system rather than those affecting programmers.

1. History

UNIX has been through many versions over the years from Edition 6 (and even earlier versions) through to the latest version, System V Release 4. However, it has not been a simple progression from one to the other, rather at times there have been many different versions available at the same time (see Figure 1.), some tailored for specific markets, others competing virtually head-to-head. For example, just a few years ago there were three main version, System III, the official AT&T product, BSD4.2, the most popular version at the time, and Xenix, the version aimed at the low end of the market.

Today, there is one standard version System V Release 4 (SVr4) (this doesn't preclude other specialised versions, such as BSD4.4 or even OSF/1). This standardisation was achieved by merging many of the best features from the different versions, including Xenix, BSD4.3, SunOS and the various research version of UNIX from AT&T Bell Laboratories.

Even more importantly, with the formation of UNIX System Laboratories (USL), and other related moves by AT&T, most vendors are making SVr4 the basis of their own UNIX versions, *e.g.* Sun's Solaris 2, Fujitsu's UXP/M and Pyramid's DC/OSx.

2. Overview of SVr4

Although SVr4 was announced a number of years ago, only in the last 12 months have commercial versions started to appear, and many more are planned for release in the near future.

In all descriptions of SVr4 much is made of the new and enhanced features, but the differences are much more than these. Users of any of the current versions of UNIX will find something that is new or modified, whether it is a System V or BSD based system. Because of this it is important not to make incorrect assumptions about the system. Even more there are many new features that have to be accounted for.

Although the changes cover all areas, some of the highlights include:

- Implementation of both the Internet and Berkeley networking utilities (*i.e.* ftp, telnet, rsh, rlogin, *etc.*) using STREAMS,
- Implementation of a virtual file system interface and a number of different file systems, including the standard System V file system (*s5*), the Berkeley fast file system (*ufs*) and a two networked file systems (*rfs* and *nfs*).

- Inclusion of symbolic links on *s5* and *ufs* file systems.
- Implementation of */proc* and */dfs*, *i.e.* the first is an interface to running procedures, especially useful for debugging, and the second an interface to currently open file descriptors. Both are implemented as virtual file systems.
- Restructuring of the system directory tree, including the addition of */var* and */stand* file systems.
- Dynamic linking and *ELF* linkage format.

3. User Differences

Depending on what system you are used to, you will either experience a major difference or else wonder what all the fuss is about. SVr4 has a number of major changes from standard SVr3.2, but many of them were common extensions, such as networking utilities and BSD compatibility.

3.1 Symbolic Links

Symbolic links are basically a file which contains the name of another file to be used in its place. Unlike normal (or hard) links they can be used to refer to normal files or directories, they can also refer to objects on other file systems, or even to non-existent objects.

These links are not without their problems. These include the getting "lost" in the file system, due to *cd <dir>* not being reversed by *cd ..* (although some shells such as */usr/bin/ksh* internally track the directories and attempt to backtrack for you). Also the links are interpreted at the point they are found, so loops are possible (although they fail after a fixed number of links), e.g. *link* → *link*.

3.2 Common Locations

Over the years the number of different locations for programs has multiplied. With SVr4 most of the common programs are located in */usr/bin*, there is, however, a symbolic link from */bin*.

This is the theory, unfortunately in practice everyone adds their own additional. For example, Fujitsu's UXP/M adds a directory */usr/uxp*, most versions of SVr4 have a */usr/lbin* and most installations will add a directory */usr/local/bin*. There is also a */usr/ucb* which is covered in the following section. Even SVr4 includes */usr/ccs/bin* which includes the C Compilation System (*i.e.* compilers, linker, *etc.*), although these programs are often linked to */usr/bin* (or if not, then the administrator should do so).

3.3 BSD Compatibility

One of the most widely publicised features is the BSD compatibility. If you are moving to SVr4 from a Berkeley compatible system then this is invaluable, on the other hand, if you are used to System V then this will have very few uses. In addition in the long run (*i.e.* over the few releases) this will be removed.

To enable most of these function you need to include */usr/ucb* in your **PATH**. To choose Berkeley functions over SVr4 you have to specify */usr/ucb* before */usr/bin* in your **PATH**. More importantly, some utilities, such as */usr/bin/sh*, change their actions depending on the order specified in the **PATH** variable.

3.4 Job Control

One other important addition for BSD is job control. This is the ability to pause jobs, and to move them from the foreground (*i.e.* the job accepting input from the terminal) to the background (*i.e.* a job running unattached to the terminal). This is a very useful feature which has long been available in BSD systems.

4. Administration Differences

The largest differences are found in the area of system administration. For some time AT&T have been enhancing the administration, both with procedures from other commercial versions of UNIX, and from the internal research versions.

New features in SVr4 include a unified approach access to the system, facilities for virtual file systems, the */stand* partition for bootable images, reorganised system partitions, new backup procedures and new

networking procedures.

4.1 File System Reorganisation

With the advent of workstations, diskless systems and other networked systems a reorganisation of the system layout is long overdue. The reorganisation has been designed with the aim of sharing files across the network. This has involved splitting files into the following directories:

root	which contains files necessary for booting the system.
/usr	which contains shareable files that are static over the life of the system. This file system can be mounted *read-only* and generally contains architecture dependent files.
/usr/share	which contains architecture independent files, *e.g. termcap* and *man* pages.
/var	which contains files and directories whose contents change over the life of the local system, such as system log files.
/home	which contains the home directories and files of the system's users.

Any of these can be mounted as a separate file system depending on the system size and requirements (and in fact some have to be *e.g. /stand*).

Along with this restructuring, various system utilities have been reorganised. Many of the utilities that were previously located in */etc* or */bin* have now been moved to */sbin* or, if they are considered non-essential, in */usr/sbin*.

Finally, the */dev* directory has also been reorganised, which related devices located in common subdirectories, *e.g.* all terminals are in */dev/term*, pseudo-terminals in */dev/pts*.

4.2 Virtual File System and vfstab

In many ways a new feature is the *virtual file system* interface and the ability to have many different file system types. Basically this is an extension of *vnodes*, introduced for *nfs*, which allows many different file system structures to share common interfaces.

Some file system types supported include:

s5	The traditional UNIX file system.
ufs	An implementation of the BSD *fast file system*, which optimises disk usage for larger files.
rfs	Remote File Sharing, *i.e.* the standard distributed file system type for System V UNIX.
nfs	An implementation of the Network File System, originally developed by Sun Microsystems.
/proc	The process file system type, which is a mechanism for accessing the address space of running processes.
bfs	A very simple file system that provides support for file-system-independent booting.

Along with the introduction of these file types there comes the problem of administration. In the case of the *virtual file systems* this is handled by specifying in the file */etc/vfstab* various information including the file system type. To handle various administrative procedures, a generalised wrapper program invokes the appropriate specific instance for a particular type. For example, */sbin/fsck* checks the file system type in */etc/vfstab* and then calls the appropriate version of *fsck* from */usr/lib/fs/<type>* directory.

4.3 Booting Procedures

Another area of major change has been in system booting. With the advent of the various file system types it would be too difficult to write a bootstrap program that handled all types. To cope with this SVr4 has introduce a simple file system (*bfs*) specifically for handling standalone programs, such as the UNIX kernel.

To handle this SVr4 has included specific types in the disk VTOC indicating which partition is of type *bfs* and thus usable by the bootstrap program.

Once the system boots you have the choice of which run-level to select (there is a way to set a default level). This has been standardised much more than previously, with the following levels.

TABLE 1. UNIX Run Level Description

Level	Description
s	Single user mode
0	Halt/Power off
1	One user mode
2	Multiuser mode
3	Distributed mode
4	Unassigned
5	System diagnostics
6	Reboot

Obviously some modes are dependent on what is possible, for example level 0 will only power off where possible while level 5 will often only halt the system, leaving it up to the user to run diagnostics. Aside from the standardisation, SVr4 also introduces a new definition for level 3, *distributed mode*. In this mode various distributed file systems are imported and exported. File systems are imported by the use of the *mount* procedure (similar to BSD systems), while the export procedures are more complicated (see below).

From the above table you will see that there are effectively two similar level, *s* and 1. The major differences are that level *s* invokes */sbin/sulogin* to login as *root* with nothing generally mounted, whereas level 1 is effectively a fully running system, but only accessible from the system console.

One final difference is that when the system changes to level 2 or 3 it also invokes the *service access facility* which controls general access to the system.

As with previous versions of System V, all these levels are controlled by the file */etc/inittab* and so these levels can be change, but it is not recommended.

4.4 Service Access Facility

With the varying accesses methods now available to the system, the previous methods such as *getty* were found to be wanting. To address the problems of no central control SVr4 introduces the *service access facility* and the related *port monitors*. The programs to control this are *sacadm* and *pmadm*.

A widely known example of a *port monitor* is *inetd*, which, although it hasn't been fully integrated, is run as one type of *port monitor*. Another, more familiar to System V users is the *listen* service, which is another *port monitor*, which has been fully integrated with the *service access facility*.

One related idea is the ability to disallow all logins if the file */etc/nologin* is present. This has been available under BSD systems but has only now been included in System V systems.

4.5 New Networking Facilities

As is widely known, the fully Internet and Berkeley networking facilities are now available, along with an implementation of the *socket* library. However, in general, these are not implemented under SVr4's *transport layer interface* (*TLI*).

For procedures using the TLI (which includes *remote procedure calls* (*RPC*)), access to various networks is control by the file */etc/netconfig*. This lists such details as the networking family (*e.g. inet*), the device to use, the access methods (*e.g.* connectionless, virtual circuit, *etc.*), and which shared libraries to use.

4.5.1 File System Export The control for exporting file systems is now handled by a number of new procedures. These include such procedures as *share*, all of which are driven by the script */etc/dfs/dfstab*. This is invoked by many programs including *mountall* and by *init* on change to run level *3*.

4.6 Network Printing

One feature in BSD systems that has long been requested has been remote printing. Along with many other changes to the SVr4 printing system the facility to do network prints is now available. This includes an emulation of BSD facilities and the ability to connect to BSD systems.

This is very simple to implement, and involves defining a remote systems using *lpsystem* and specifying those systems as the destination, rather than a device. To accept remote printer requests the SVr4 *listen* facility is used, with a request to also listen on the BSD printer port allowing connections from BSD systems. One point with this, for systems with multiple network connections, each need a separate entry.

One other problem at present is that there still appear to be many implementation problems with most implementations.

4.7 Backup Procedures

Another area that has undergone considerable reworking is the backup facilities. SVr4 implements a "unified" approach to many different backup procedures, from a file by file basis, to a full partition image, to a disk image. These are controlled by a set of routines that allow backups to be scheduled and run automatically or run under operator control. They also optionally keep detailed logs of the files backed up and have commands to move these to other locations. Further, there are now commands for users to request restores of their own files and directories, as well as the administrator to restore partitions or disks.

The command to run a registered backup is *backup*, which may then invoke *bkoper* to communicate with an operator and also invokes various other utilities to perform the backup. The programs to perform the physical backup are all located in */etc/bkup/method* and have specific procedures for communicating with *backup*. Unfortunately while the new programs are good (although they still have some bugs as would be expected in a new system), the backup formats are still the traditional ones, *cpio*, *volcopy* and *dd*. There is still nothing that can handle file deletion or recreate individual files that are missing blocks (as is found with many databases).

On a positive note, however, BSD's *dump* and *restore* are also supported (called *ufsdump* and *ufsrestore*), but only on *ufs* file systems.

4.8 Security

Although there has been an increased level of security awareness over the years, some of which is reflected in SVr4, there are still problems to be overcome.

SVr4 implements a shadow password file, password aging, account expiry and detailed password checking. However, many of the system changes increase the need to override these security procedures. For example the printer scheduler *lpsched* now runs as *root*, similarly because of the use of file system permission in the */proc* file system *ps* and related programs now require to be setuid *root*.

5. Conclusion

The advertising people claimed some years ago *System V - Consider it Standard*, this is now coming true. Most vendors are implementing versions of it and it is by far the dominant version in the market at present. It is even so standard that many of the bugs are common across all platforms, *e.g.* the TZ variable does not handle Australian daylight savings under UXP/M or Interactive.

Despite these initial problems, SVr4 is an exciting development, it provides most of the functionality that users have been asking for for some time and yet provides many new features and "goodies" to keep administrators busy for some time.

6. References

[1] UNIX System Laboratories (1990): *UNIX System V Release 4 Migration Guide*, Prentice Hall, ISBN 0-13-933821-7.

[2] Leffler, S.J., McKusick, M.K., Karels, M.J. and Quarterman, J.S. (1989): *The Design and Implementation of the 4.3BSD UNIX Operating System*, Addison-Wesley, ISBN 0-201-06196-1.

A Filesystem for a Multi Gigabyte Jukebox

Rex di Bona
Ray Loyzaga

The University of Sydney

ABSTRACT

Data Storage Devices are rapidly increasing in capacity. This has created problems in both backup and archival of data stored on these devices. With the advent of cheap, removable, optical storage devices, and jukebox control units it is now possible to have many gigabytes of archived or backed up data online.

1. The Disk Storage Problem

Each system administrator has to cope with the problem of long term disk storage. It is a well known problem that users accumulate, in fact almost aggregate, files. Each file so important that it could not be trusted to such a flimsy mechanism as tape.

Instant access to each of these files is also required. A user will desire instant access to any datum that they have collected, and to search through tape is a long and arduous process. This, while being perhaps slightly facetious is an accurate outline of a user who is not charged for their storage use. A further advantage is the reduction in human interaction to obtain tapes, and less wear and tear on tapes.

The Basser Department of Computer Science uses, at last count, 11 Machines with 15 Gigabytes of online storage for the academic users. Some of these machines are dedicated to specific projects, whilst others are available to the general academic populace. The overwhelming bulk of the data stored on the disks is archival, but *required for immediate use*. It must be stated that a lot of the disk, actually several gigabytes, is sources to various systems, including X11R4, which we currently use, and X11R5 which we are moving to.

We are trying to archive most, if not all, of the archivable material, but onto a medium more accessible than magnetic tape. To achieve this purpose the department used a Hewlett-Packard HP6300 Jukebox kindly donated by Hewlett-Packard for this project. The jukebox is a SCSI device that can hold up to 20Gb of data online as 32 disks each holding 600Mb, and two drives, each capable of reading and writing one side of a disk. Onto these disks the archival material will be stored, and older disks will be removed from the jukebox as the device fills up.

As a point of terminology each disk is thought of as containing two platters, an up platter, and a down platter. This is because to read the down platter the disk has to be physically flipped by the autochanger.

2. The File System

The jukebox presents an interface similar to that of two normal disks. Each disk drive is an independent unit, capable of reading and writing to a single platter. The platters are moved by an autochanger mechanism which is a separate SCSI device. The controlling machine is responsible for co-ordination between the drives and the autochanger.

Because the jukebox was to be a system wide resource, and because the Department has considerable experience with building user mode NFS filesystems it was decided that the jukebox would be presented to the kernels of the machines as a single NFS filesystem. The filesystem would appear to be the sum of the sizes of each of the platters. It was originally envisaged that there would be one jukebox wide filesystem, but for testing purposes the current implementation has a limitation that files cannot cross platter boundaries.

A more severe problem with attempting to have one filesystem span the entire jukebox is the problem of addressing. A thirty two bit pointer is insufficient to address individually each byte of storage on the jukebox. Either files would be limited to four gigabytes, or to the size of a platter, and the easiest solution

was to limit files to the size of a platter.

2.1 An NFS Protocol Filesystem

As each of the machines in the department could mount NFS file systems, and the jukebox was to be a department wide resource, it was decide to make the jukebox controlling software implement the NFS protocol. The software to run the jukebox was also designed to be as system independent as possible, and to require minimal kernel changes to function. This led, along with the experience in the department with user mode NFS systems, to the jukebox controlling software being a user mode NFS server. The software is run as an ordinary user, and requires no kernel support for the NFS protocol interface.

This led to the software being easily debuggable by an ordinary debugger, and changes to the software did not require a kernel rebuild and machine reboot. To improve the ability to debug the initial software each platter has a self contained filesystem on it, a filesystem that the standard utilities, *fsck*, etc would work on. This meant that a large body of code didn't have to be written especially for the jukebox file system and improved the development cycle. As there was a real system supported filesystem on each platter the routines provided by the kernel for manipulating filesystems could be used to manipulate files on the individual platters.

2.1.1 The File Handle One draw back in using a standard filesystem and in using the kernel to manipulate the files is the problem of a file handle. In the NFS protocol files are identified by file handles, opaque 32 byte structures that can be passed to a file server to access a file, or directory. Because NFS is stateless a file handle must be unique across reboots, so some sort of internal data structure cannot be used as it would be lost across reboots. For kernel bound NFS servers this is not a problem as an inode can be used as the file handle. The kernel can manipulate files given just their inodes, but a user level process cannot, it can only manipulate open files or files specified by pathnames.

To solve this a filehandle was devised that could be used to map onto a filename in a one to one mapping. Each file handle would map onto a single file and would be constant across reboots. The file handle thus devised contained the platter identifier, the inode of the file (or directory), and the inode of the parent directory. Given these pieces of information the only system dependent routine was created, *getp()* took the file handle and returned the path to the file specified by that handle. It did this by reading the blocks associated with the parent directory inode, and storing the inode of its parent and the name of either the file or the child directory. Doing this recursively the algorithm would produce the full pathname for the file working from the file to the root inode of the platter. When the routine reached the root it would terminate. This 180 line routine needed to know the internal structure of directories on the platter, the only routine in the server that needed to know any internal structure of the file system.

2.1.2 NFS Problems Chosing NFS was not without its problems, the filehandle was a problem, and other, more interesting problems were found as development progressed. The NFS specification defines several structures as being opaque, their values were not to be interpreted by the client machine. It was found that the NFS reference implementation implemented by manufacturers did indeed look *inside* these structures and interpret the values therein. This caused some problems as our implementation didn't use these structures for the same purpose as the reference implementation.

One of the more intriguing problems was that the length of a directory was supposedly held inside an opaque structure, and was used on an end of directory message to represent the size of the directory. We left this value as zero on this message which resulted in zero length directories, and even more worrisome, each file in the directory would appear twice. This was because the client kernel would receive the first return message which contained all the entries and had the end of directory flag set. The client kernel would ignore the end of directory flag, request more information with the start pointer as the first entry, which resulted in a duplicate set of entries being sent.

For large directories we had the opaque pointer being a pointer to an internal data structure, so we ended up having directories being several megabytes in length. This problem was solved, but required more work to implement. As the supposedly opaque data structure was not so opaque a mapping has to be kept in the server with *offset in the directory* to *position in the data structure* information.

A more serious problem was the failure of the NFS clients to include user or mode information in create requests. A create request supposedly had two fields that contained the uid and mode to create the file or directory with, but these were sent as zero and 07777 respectively, resulting in publically writeable, set-uid root files, a grave security risk. The information could be obtained for the user from the credentials with the request, and it was determined that a mode of 07777 was the *don't set* mode, instead of -1 as the standard stated.

2.2 The Root Directory

The root directory is an imaginary directory. It holds an entry for each visible platter, and two special files, `ctl`, and `status`. An example is shown in Figure 1. This directory is created when the server is either started or reinitialised.

```
joyce # ls -lisa /jukebox
total 49
     2    1 drwxr-xr-x   2 root     root          512 Mar 11 20:08 .
     2    3 drwxr-xr-x  27 bin      bin          1536 Mar 11 03:00 ..
   100    1 drwxr-xr-x   2 root     root          512 Mar 11 20:08 0000000
   101    1 drwxr-xr-x   2 root     root          512 Mar 11 20:08 0000001
   102    1 drwxr-xr-x   2 root     root          512 Mar 11 20:08 0000002
   103    1 drwxr-xr-x   2 root     root          512 Mar 11 20:08 0000003
   104    1 drwxr-xr-x   2 root     root          512 Mar 11 20:08 0000004
   105    1 drwxr-xr-x   2 root     root          512 Mar 11 20:08 0000005
   106    1 drwxr-xr-x   2 root     root          512 Mar 11 20:08 0000006
   107    1 drwxr-xr-x   2 root     root          512 Mar 11 20:08 0000007
   108    1 drwxr-xr-x   2 root     root          512 Mar 11 20:08 0000008
   109    1 drwxr-xr-x   2 root     root          512 Mar 11 20:08 0000009
   110    1 drwxr-xr-x   2 root     root          512 Mar 11 20:08 0000010
   111    1 drwxr-xr-x   2 root     root          512 Mar 11 20:08 0000011
     3    0 --w-------   2 root     root            0 Mar 11 20:08 ctl
     4   33 -r--r--r--   2 root     root        16384 Mar 11 20:17 status
joyce #
```

Figure 1. A Root Directory

Each platter is identified by a unique platter number, given to it at platter creation time. When the server initialises this directory it firstly creates the `ctl` and `status` files, then it scans all the resident platters and obtains their platter numbers. This directory is unique in the jukebox server in as much as it is the only directory that cannot be modified, even by root. Any attempt to modify this directory, to chmod a file, to create a file, or remove a file, will result in a permission denied message. The date of the entries, apart from the status entry, do not change, and are the time when the server was started. The status entry changes continuously so that an attempt to read it results in an actual read, not the cached copy from the local NFS client.

2.2.1 Platter Directories Each platter has its own file system, mapped into the jukebox hierarchy starting at the platter number. To the NFS client it appears that a seamless transition is made between the root directory and the root of each platter, so doing a listing of the platter root directory doesn't result in any unusual inode numbers for the `.`, or `..` entries.

```
joyce # ls -lisa /jukebox/0000001
total 20
    101     1 drwxr-xr-x   2 root     root          512 Mar 11 20:08 .
      2     1 drwxr-xr-x   2 root     root          512 Mar 11 20:08 ..
2097156     0 -rw-------   1 root     root            0 Jan 29 15:01 0000001
2097155    17 drwxr-xr-x   2 root     root         8192 Jan 20 18:16 lost+found
2097280     1 drwxr-xr-x   3 root     root          512 Jan 28 15:05 pgrad
joyce #
```

Figure 2. A Root Directory on a Platter

The inode numbers returned by NFS are unique across a filesystem, so the real disk inode number could not be used as the returned inode number (properly called fileid in NFS terminology). Instead the inode number on each disk is prepended with a platter number set sufficiently high that duplicates will not occur. This results in extraordinarily large inode numbers as can be seen in Figure 2.

2.2.2 Control Files The two control files `ctl` and `status` are ASCII files similar to those implemented in the Plan 9 operating system. `Ctl` is a write only file that accepts control messages, and `status` is a read only file that describes, in a fixed format, the current status of the jukebox and platters. We shall present the actions of these files later.

The server will respond to a file system stat request by returning the sum of available and used data, which interestingly enough overflows the `df` command's calculations, resulting in interesting statistics on the jukebox as shown in Figure 3.

```
joyce # df /jukebox
Filesystem                Type  kbytes     use   avail %use  Mounted on
joyce:/jukebox             nfs -948358       0 -962068   0%  /jukebox
joyce #
```

Figure 3. Df output for the Jukebox

2.3 The Platters

Each platter is formatted and set out as a single filesystem. This was originally to allow for the use of the standard filesystem tools to interact with the platters, tools that would have to be rewritten if a different filesystem format was picked. It also allowed for the kernel routines to be used to access the data stored on a platter. This causes a problem however, it is now impossible using the current implementation to create a file that is greater than platter in size, and a file cannot span more than one platter. This results in the situation where even though there is free space on the jukebox a file cannot grow as its platter is full. We will discuss ways that this problem can be circumvented, but we still have a hard limit on file size of four gigabytes imposed by NFS.

2.3.1 Changes for a Single Filesystem There are two ways we can allow for files to grow larger than a single platter, we can either redefine the platter filesystem format to a format that allows a file to be spread over multiple platters or we can arrange for an additional layer of software, above the current jukebox software, to handle the problem, and arrange for a file to be split as necessary.

The idea of redefining the format of the filesystem was played with for a while, and may be implemented in future work, but would require the creation of the equivalent of newfs/mkfs, fsck, dump, restore, and a lot of the kernel routines. This is a worthwhile research project, but not necessary for our use.

Since the jukebox was to be used for an archive server it was noted that most of the data on the system would be static. Once on the jukebox it would neither grow nor shrink, but would just be read. If a data file was desired to grow the file would be moved from the jukebox back to fast disk, and manipulated there. When the file was unreferenced for a suitable amount of time it would be moved back to the jukebox, but not necessarily in its original position. Since we know the size of a file when it is being moved to the jukebox we can move it to the platter that has sufficient free room, or if necessary we can have the last file on a platter overlap onto a new platter. This would require only one file per platter extending across to a new platter, a simple enough special case.

This idea of only allowing one file to grow across a platter boundary has not been implemented currently, but will be incorporated in the version of software under development. This will allow files to grow to the maximum size allowed by the NFS protocol.

2.4 Conclusion

The filesystem presented above has been implemented and is currently running on a MIPS Magnum RC3330 computer. It uses 350Kb of virtual memory, including the caches that will be described shortly. It achieves a throughput of 300K reading, and about 100K writing, due to the speed of the optical disks. The source is just under 4500 lines of C code with that being 1600 lines of NFS interface code, 1200 lines of file handling code, 500 lines for both the filehandle and file caches 190 lines for the filesystem dependent code, and 1000 lines for both the jukebox autochanger control and the pseudo root directory with ctl and status files.

The system performs quickly, and can recover from most soft errors well. Its worse fall back is to do an automatic rescan of all the platters and reinitialise all its internal tables. This action is lengthy, taking about 15 seconds per platter, so for a fully populated jukebox it takes about 8 minutes to reinitialise.

The main problem encountered has been that the server code is single threaded. This means that operations that require a platter change hold up operations that do not need to be delayed. This is one area that needs more work, but for an archive device a delay of about 16 seconds for any file was deemed satisfactory performance.

3. The Caches

Since it takes a considerable amount of time to change platters it was imperative that caching of various things occur to improve performance. The three things that the jukebox server uses are, the mapping from filehandle to pathname, the *stat*'ing of a file, and the reading and writing of a file. All three of these were cached to improve performance. The main objective of each cache was to reduce, or eliminate the number of platter moves that occurred.

3.1 The Filehandle Cache

The filehandle cache keeps a cache of filehandles to pathnames. Each time a filehandle is created or looked up using the *getp()* routine the pathname that corresponds to that filehandle is stored along with the filehandle in a thousand element LRU cache. This cache is indexed by filehandle only.

3.2 The Stat Cache

The most common NFS operation is a file lookup. This takes a file handle for a directory, a file name for a file in that directory, and returns the filehandle for the specified file and a stat structure for the file. To allow for this the filehandle cache was extended to have the stat structure held along with each filehandle, and with each directory a list of the files in the directory was optionally held.

The list of files stores the name of each file and the inode number for that file. This allowed a lookup operation to operate entirely within the cache, as the first filehandle would be used to obtain the structure for the directory, the list of files would be traversed looking for the specified file and if not found an error returned. If found the inode number for the file was taken with the inode number for the directory and the platter id for the directory, both obtainable from the filehandle for the directory itself, and combined to form the filehandle for the file. This was then used to search the cache and if found the information returned. If the cache was hit both times the platter containing the files wouldn't be accessed at all.

3.3 The File Cache

To speed up read accesses to the jukebox a write through cache of files was created. This cache will cache up to the first three megabytes of a file on a fast disk. The cache is accessed on full jukebox pathname. The cache is write through in case of unexpected system shutdown or platter removal. The jukebox server software doesn't keep a list of files cached in the file cache, when a request comes in for a read or write the server will create the file if necessary.

There is a problem with sparse files, and files containing blocks of zeros in the cache. The NFS protocol allows out of order writes, so a file in the file cache may be sparse even though data exists in the holes in the file. To account for this problem the server checks each block read from a cached file. If a block reads as all zeros then the actual block from the real disk is read. This is because it is not possible to tell whether the block was all zeros or was a hole in the cached file.

The server only creates files in the file cache. A separate process is used to trim the size of the cache. It is run infrequently, and removes all files not accessed in a certain period. If the cache fills up then the performance of the server is impaired, but the correct data is still returned.

3.4 Rebooting

When rebooting the jukebox server machine it is preferred that both a shutdown and an unmount message be sent to the jukebox server. This ensures that all filesystems are consistent and that the current layout of the jukebox platters in their storage slots is saved into a configuration file.

When the jukebox server starts up it checks for a configuration file, and if present initialises itself from this file instead of from the jukebox itself. If on any access after that a discrepancy is found between the information in the configuration file and the actual jukebox layout the server will automatically reinitialise itself. For fully populated jukeboxes the configuration file reduces start up time drastically. If the server crashes unexpectedly, and the old configuration file still exists the old configuration file can be used for the jukebox server startup. This is possible as the jukebox server tries to keep platters in the same slots whenever possible which means that the configuration file stays accurate. This slightly increases the work that autochanger has to perform, but the increase is a marginal one, being a disk flip on occasions.

4. Kernel Modifications

The jukebox server is implemented on a RISC/os 4.52 kernel with additional SCSI commands for controlling the autochanger, and some additional IOCTLs for manipulating the platter header information. These changes were added to the SCSI driver and the kernel rebuilt. The additional commands for the autochanger were implemented as IOCTLs on the raw disk device. Most of the changes to the kernel were because the kernel assumed that all devices were either tapes or disks, and that all disks were fixed. As removable media become more prevalent the need for these types of changes to the kernel will be reduced.

One kernel change discarded as too costly was to modify the file type of archived files to a special archive type. This was discarded as it would require modifications to every kernel that would want to use the jukebox system. Whilst this method would give the nicest user interface the amount of work for system administrators would be prohibitive, and would delay the adoption of new operating systems at sites until the appropriate modifications were done. As the jukebox system currently stands only the server machine requires kernel modifications, all other machines access the jukebox through NFS, and the kernel changes are only due to deficiencies in the kernel which should be corrected soon (hopefully)!

4.1 Changes to the SCSI Driver

The changes to the SCSI driver included adding the SCSI commands to make the autochanger move, to obtain information about the number of disks, the layout of the disks, the layout of the disk drives in the jukebox. It is expected that these commands will be standard in kernels soon, as they are all part of the SCSI standard.

4.2 Additional IOCTLs

The IOCTLs added to the system were those necessary to implement the additional SCSI commands, and one to make the system reinitialise its volume information about a fixed disk. This IOCTL was required to correct the assumption that all disks were fixed disks and as such couldn't be removed, this resulted in the partition information not being read after a platter exchange.

5. *The Control Interface*

5.1 The Status File

Figure 4 shows the output from the status file. It is a fixed format ASCII file, and is available on any host that the jukebox is mounted on, which allows for control of the jukebox to be done from a remote host.

```
joyce # cat /jukebox/status
Jukebox Status
Transport Elements     : 1 (0 to 0)
Storage Elements       : 32 (11 to 42)
Input/Output Elements  : 1 (10 to 10)
Data Transfer Elements: 2 (1 to 2)
Disk Drive 0 (SCSI 4L0) holds: Active: 0000001 (295086Kb, 281470Kb avail)
                               Inactive: 0000000
Disk Drive 1 (SCSI 5L0) empty
Storage Element   0 holds: Up: 0000001 (295086Kb, 281470Kb avail)
                         Down: 0000000
Storage Element   1 holds: Up: 0000002 (295086Kb, 295077Kb avail)
                         Down: 0000003 (295086Kb, 295077Kb avail)
Storage Element   2 holds: Up: 0000007 (295086Kb, 295077Kb avail)
                         Down: 0000006 (295086Kb, 295077Kb avail)
Storage Element   4 holds: Up: 0000008 (295086Kb, 295077Kb avail)
                         Down: 0000009 (295086Kb, 295077Kb avail)
Storage Element   5 holds: Up: 0000011 (295086Kb, 295075Kb avail)
                         Down: 0000010 (295086Kb, 295075Kb avail)
Storage Element   6 holds: Up: 0000004 (295086Kb, 295077Kb avail)
                         Down: 0000005 (295086Kb, 295077Kb avail)
joyce #
```

Figure 4. Example Output from the Status File

The platter `0000000` doesn't have associated with it the actual disk values for size of platter, and free space available. This is because that filesystem was not unmounted correctly due to a system crash and is still dirty. The jukebox software recognises that a filesystem is on that platter, but will not clean it until necessary. Manual intervention could be done by locking drive zero and manually `fsck`'ing the platter. The status file holds information about the entire system, number of drives, mailslots, storage slots, and autochanger arms. It also contains dynamic information, the platters that are mounted in each drive, the home storage spaces for each platter, and the space available on each platter.

5.2 The CTL File

The commands that the control file accepts are presented in Table 1. These commands are sent as ASCII strings, so the echo command can be used to control the jukebox's functions.

Command	Action
dump	Dump out the internal caches and structures (Used for debugging).
lock	Lock a drive and remove it from jukebox access.
noremove	Disallow front panel control of the jukebox. This is a security measure to ensure the software's idea of the state of the physical jukebox corresponds to reality.
remove	Allow front panel control of the jukebox.
rescan	Reinitialise the internal idea of where disks are.
shutdown	Write out the current configuration of the jukebox.
unlock	Allow a drive previously locked to be used by the jukebox again.
unmount	Remove the platters from the drives and return them to their storage spaces. This is used in preparation for an orderly shutdown of the system.

TABLE 1. Commands Accepted by the Jukebox Server

6. Other Uses for the Jukebox

The jukebox as shown is used as an archive server. It could however be used for other purposes. One possible use for the jukebox is as a backup device. Each night a script would run that made copies of changed files to the jukebox, and then during the day these changed files could be backed up from the jukebox to tape. The files also remain on the jukebox to provide a first level user accessible backup store. When Basser had spare disk space a copy of files were kept in an online backup directory called /backup for each device, and copies made in there each night. This proved remarkably effective with most backup requests able to be satisfied from the /backup directory.

7. Conclusions

The removable media jukebox provides an alternative to tape for archival of infrequently used data. It provides the random access patterns of regular disk, with a slight loss in speed, but provides the ability to have much greater quantities of data available. This paper has presented one method where this storage medium was used to provide archival of this infrequent data at low cost to the department. Overall the performance of the jukebox has been more than satisfactory, with data availability being much higher, and faster, than with tape storage.

AARNet

Engineering a Connection to AARNet

Peter Elford
Network Coordinator
Australian Academic and Research Network

AARNet

History

Management Structure

Architecture

AARNet Membership

Mail Affiliate Membership

Network Affiliate Membership

AARNet

HISTORY

% In the beginning ... two major network infrastructures

- ACSnet (Australian Computer Science Network)

- X.25 Networks (INFOPSI, Coloured Book)

% driven by interest groups: no focus on the provision of networking services to the entire higher education and research sector

% this requirement was identified by the Australian Vice Chancellors Committee (AVCC) in mid 1988

% network architecture was agreed upon at the the 1988 Networkshop; ACSnet and X.25 models not adopted

% Network Manager was appointed in 1989; securing of funds and purchase of equipment took 12 months

% Network Coordinator appointed in January 1990

% Installation commenced in May 1990 - lasted 6 weeks

% rapid growth in membership and usage

AARNet

MANAGEMENT STRUCTURE

%AARNet is owned and operated by the AVCC (in partnership with the CSIRO)

%A service to further the aims and objectives of the AVCC member institutions (universities) and CSIRO

% Supported by

- Technical Group within the AVCC (staff of 2)
- Operational contracts with regional hub sites
- Technical Working Groups

% Policy through AARNet Advisory Board

%JFunded largely by the member Universities, with contributions from the Australian Research Council and the CSIRO

% Total 1992 budget approximately $2.8M ($2M of which is paid to AOTC)

AARNet

ARCHITECTURE

%AARNet is a network of networks

%JNot a single computer system, or even several computer systems (there is no AARNet userid)

%AARNet interconnects the Local Area Networks of each member institution and organisation with multi-protocol routers

%JDominant protocol in use is TCP/IP (because that is what is deployed within the academic and research sector)

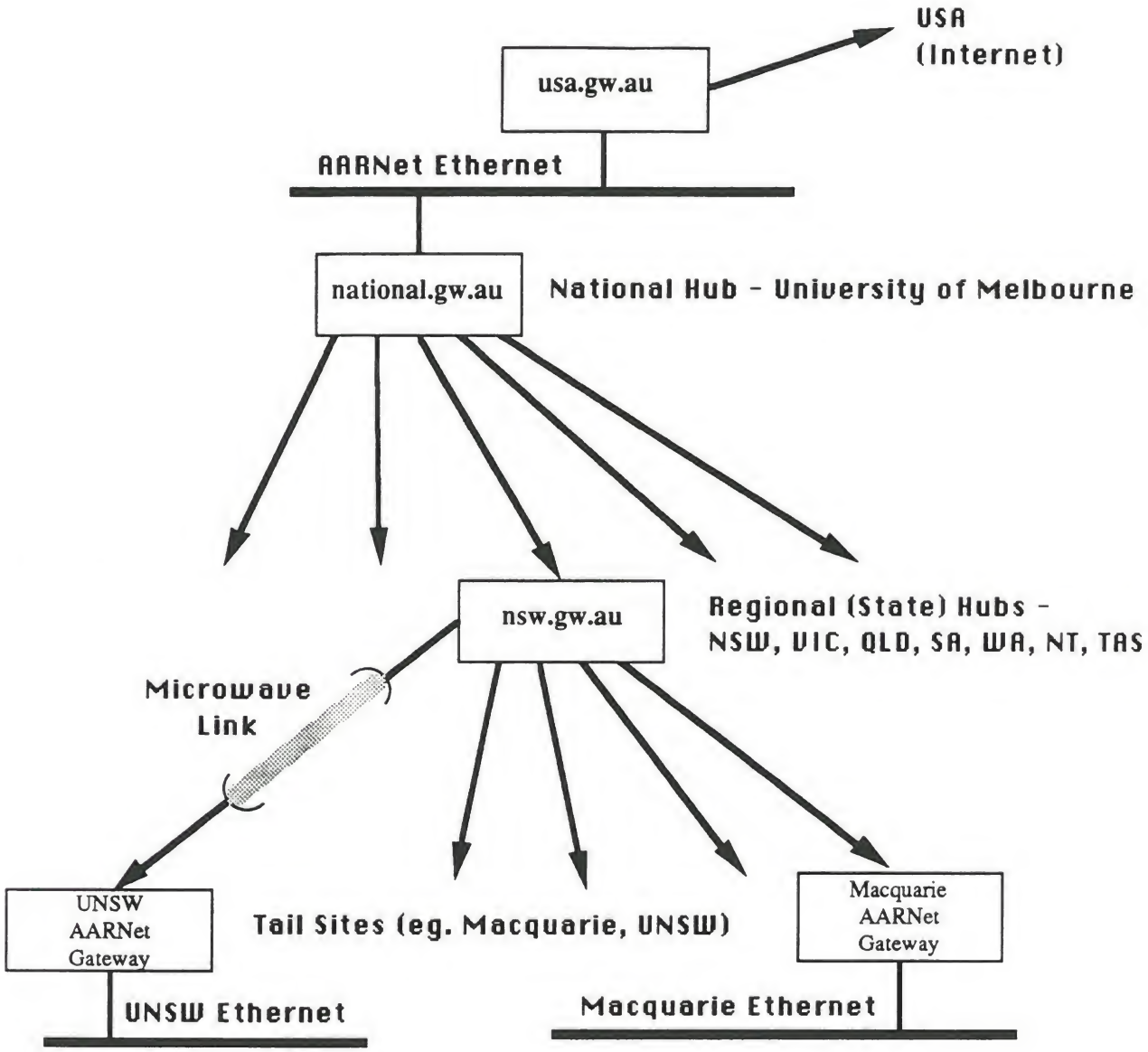
% Provides transparent transport from one campus to any other (and the world)

% Services available are those provided by the systems connected to the campus LANs

% Part of the Global Internet

AARNet

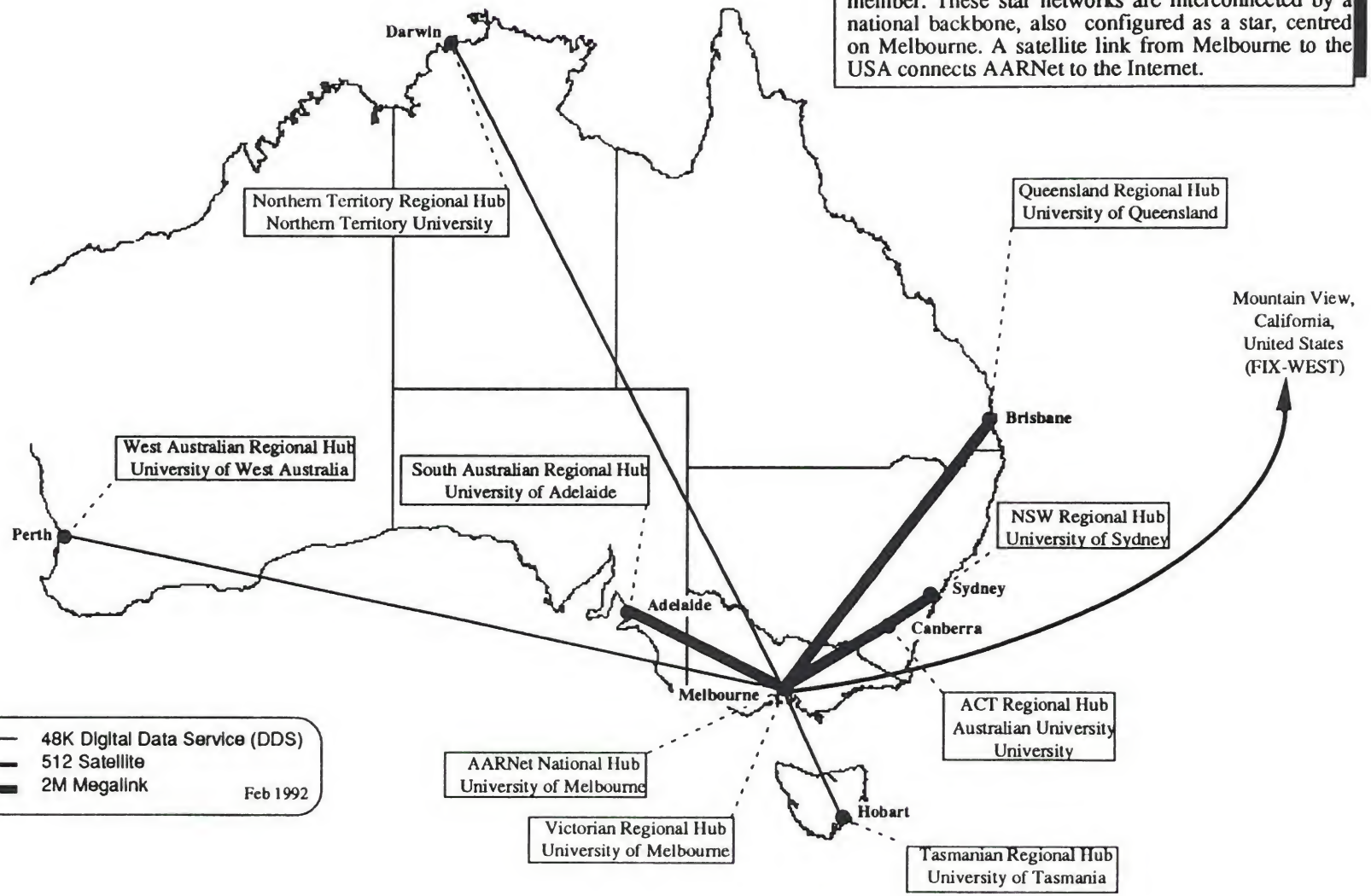
ARCHITECTURE



Australian Academic and Research Network

AARNet

Each regional state network consists of a single hub router with 48K or microwave links to each AARNet member. These star networks are interconnected by a national backbone, also configured as a star, centred on Melbourne. A satellite link from Melbourne to the USA connects AARNet to the Internet.



AARNet

AARNet Membership

% All 38 Universities (minimum connection speed 48K)

% CSIRO Divisions in all states (9.6K - 10M)

% Libraries (UNILINC, National Library of Australia)

% Total number of connections to AARNet (including affiliate members) in excess of 31,000

% Peer network with other IP research networks throughout the world that make up the Internet

% Access via electronic-mail gateways to other networks such as BITNET, JANET and the global UUCP networks

% Provides ability to communicate with/access:

- peer researchers
- supercomputers
- on-line information services (libraries, WAIS)
- vendors
- vast amounts of public domain software
- much, much more!

AARNet

AARNet Membership

% The AVCC extends membership of AARNet to other organisations under the Affiliate membership program

1. To support collaborative undertakings with the higher education and research sector;

2. To enable and/or improve the provision of services to the higher education and research sector;

3. To facilitate the dissemination of information through Australia in support of the activities of the higher education institutions and the CSIRO;

4. To encourage broad participation in the development of appropriate technologies to support the current and future requirements of the national academic and research endeavour.

% Affiliate Membership is offered in two different ways:

Mail Affiliate Membership

Network Affiliate Membership

AARNet

AARNet Mail Affiliate Membership

% Enables an organisation to exchange only electronic mail with AARNet and it's connected networks

% Does not require a permanent connection to AARNet

% Annual Membership fee of $1,000 buys

- registration of an electronic mail organisation name
- transmission of an unlimited volume of electronic mail over AARNet facilities

% Registration is in the form of an Internet Domain Name System Mail Exchanger (MX) record. This global record directs electronic mail to a specified gateway system, eg.

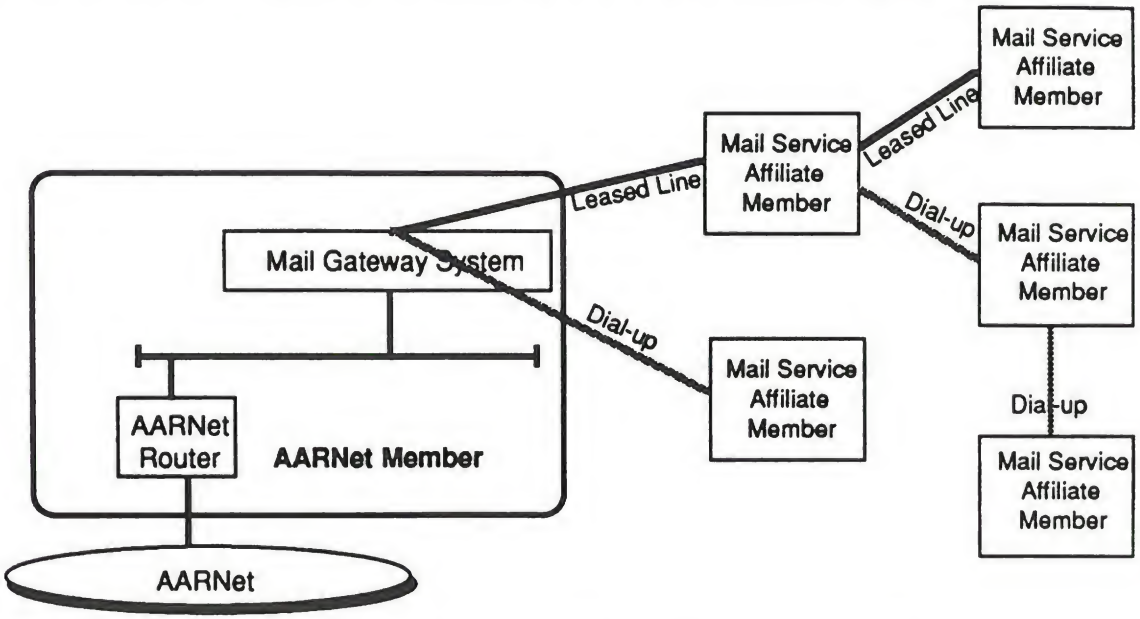
anbg.gov.au IN MX spider.ento.csiro.au

% Mail affiliate organisations must make their own arrangements with the gateway site to

a) spool their mail until it can be collected
b) provide a means by which spooled mail can be retrieved, and outgoing mail sent

AARNet

AARNet Mail Affiliate Membership



% "Mail Feeds" can be anything that works:

- SUN III (ACSnet)
- MHSnet (from MHS Systems Pty Ltd)
- UUCP (UNIX to UNIX Copy Program)
- Dial-Up terminal to host system
- Private TCP/IP (not FULL network access)
- PhoneNet (VAX/VMS systems)
- Proprietary solutions such as QuickMail

% Choice driven by systems in use and their configuration at the affiliate members site

AARNet

AARNet Network Affiliate Membership

% Provides a full network layer (TCP/IP) connection of the affiliate member's network to AARNet and hence the Internet

% Full range of IP applications are available

- electronic mail (SMTP)
- file transfer (FTP)
- remote login (TELNET)
- remote windows (X)
- many others

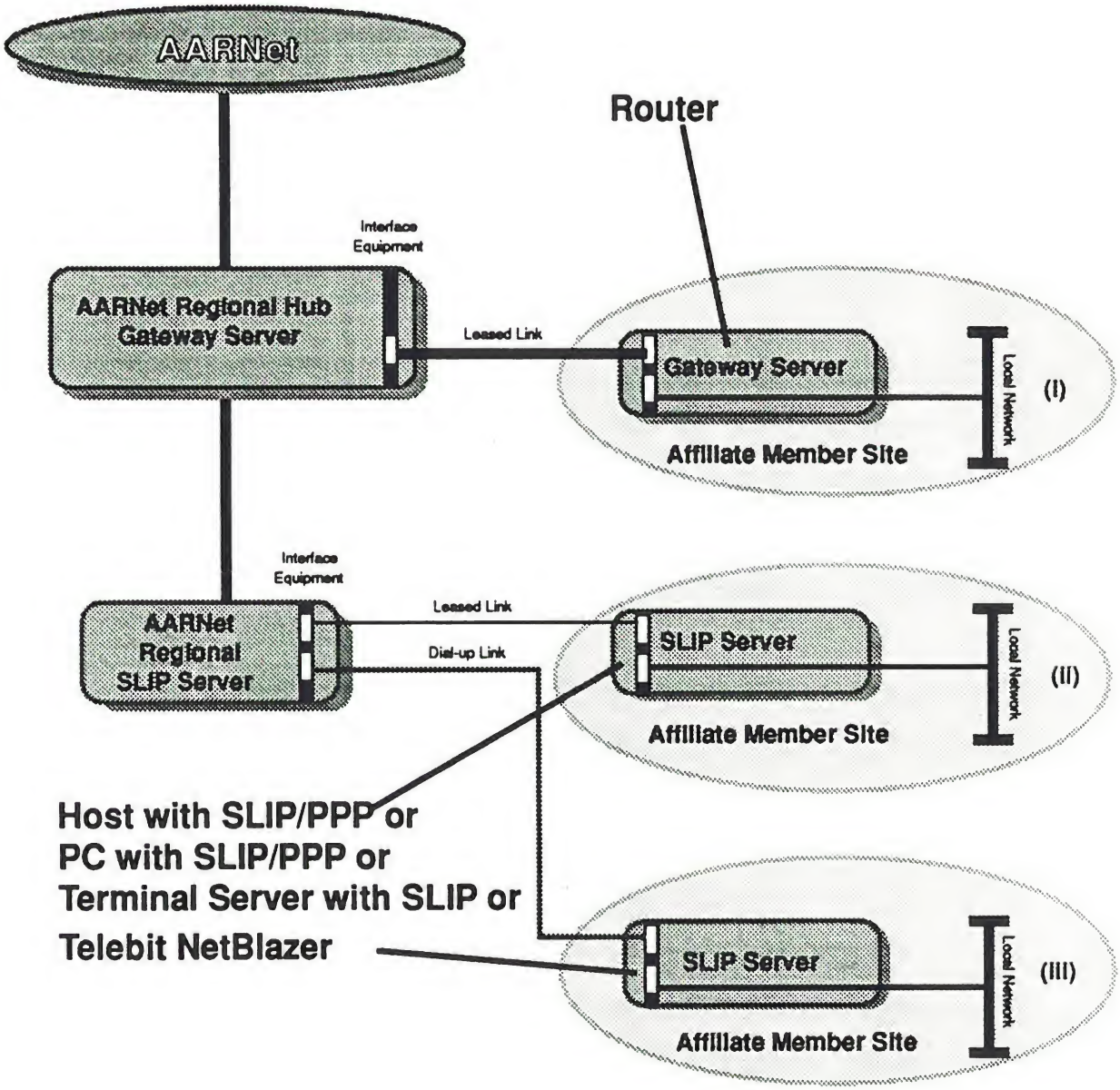
% annual membership fee based on the speed of the TCP/IP link between AARNet and affiliate member

9.6K (or less) dial-up	$5,000
9.6K leased line	$8,000
19.2K leased line	$15,000
48K leased line	$25,000
64K leased line	$30,000

% affiliate must also pay cost of equipment to terminate link, as well as associated Telecom charges

AARNet

AARNet Network Affiliate Membership



Preferred Connection Methods to AARNet

AARNet

AARNet Network Affiliate Membership

% Connection fees to connect to AARNet servers

Router	$3,500	(synch, high speed)
SLIP/PPP Server	$1,000	(aysnch, low speed)

% Configuration:

- Must use IP network numbers allocated by US Network Information Centre (NIC) and advise AARNet what these numbers are

- Must install IP software (or appropriate gateway software) on systems that are to have AARNet access

- Must register an organisation Internet domain name and operate an Internet domain name server

- Must ensure IP routing is (correctly) reticulated throughout local network

- Must configure electronic mail software

% Security
- best security achieved through router connections

AARNet

AARNet Affiliate Membership Summary

% Mail Only

($1,000 + Gateway Charge + Phone Calls) per year

[+ Software] [+ Modems] once off

% Network

$F(bandwidth) per year

[+ Connection Fee] [+ Router/SLIP Server]
[+ Modems] [+ Software] once off

%Alternatives

- Telecom Keylink (individuals)
A gateway exists between AARNet and Keylink
(C:AU, A:TELEMEMO, P:oz.au, "RFC-822":"<pte@aarnet.edu.au>")

- Corporate Gateways - Companies on the Internet
eg. Digital, Sun, IBM, Amdahl, ...

% Questions:P.Elford@aarnet.edu.au
Phone: (06) 249 3542 Fax: (06) 249 1369

System administration made easy

Peter Gray

March, 1992

1 Abstract

A method is described whereby NIS (Network Information Services) maps are used to control access to a set of or all UNIX machines on a network. This allows the machines to share a single password file and file system (via NFS) while at the same time allowing the system administrator to designate machines as available only to selected groups of individuals. This allows a single file to control access to all machines on a network. Access control is by group names or user name and is based on regular expressions.

The system also incorporates BSD style resource quotas again controlled by a single NIS map. Quotas are allocated by user name or group names and different quota sets may be used for different classes of machines, e.g. servers or workstations. Disk quotas are also included.

The system uses a "fake" login shell which passes control to the real shell (in our case the Bourne shell) upon completion. The system makes extensive use of a shared library so that alterations or additions to the control logic can be made without the need to update copies of the "fake" shell.

2 Introduction

The Department of Computer Science at the University of Wollongong faces the same problem as all academic units in today's political and economic climate, i.e. steadily decreasing resources trying to cope with increased workload. In particular, increasing number of UNIX machines (both servers and workstations) with no increase in system administration personnel.

For this reason the department has invested considerable time and effort into trying to reduce the system administration requirements of our UNIX hosts while at the same offering a better computing environment to our users.

This paper introduces one of the aspects of the system that developed, namely the access and quota control mechanisms.

3 requirements

The major requirements of the systems we developed are listed below.

3.1 A uniform environment

We wanted all the machines administered by the department to offer the same computing environment to the user. This means the user will have one account (and one password) which is valid on all machines. He/she will have a single home directory and a single mailbox. All commands will be available on all machines and function identically.

This requirement could only be met because all departmental UNIX machines are SPARC based and run SUNOS.

It would be possible to offer a reasonable approximation to a uniform environment in a heterogeneous network but would certainly be more difficult.

3.2 Ability to control machine access

Although we wanted a single password file for all our machines we also wanted to be able to restrict access to machines based on the user name and/or the groups of the user. This was to enable us to designate machines for particular purposes. For example, a machine could be designated for staff use only. This was particularly important because the department administers all the UNIX based machines for the faculty and we wanted to ensure that departmental machines were not subjected to increased load from mathematics or engineering students.

3.3 Resource control

The system had to provide us with a mechanism to allocate resource limits, both for disk and CPU/memory. The quota system had to allow for limits to vary depending on the machine being used and the values of the limits to be selected based on user name or groups. We debated whether the system should allow quotas to vary depending on time of day but decided the extra functionality would not be required[1].

3.4 Centralized configuration files

All information for controlling the system had to reside in a small number of master files kept on a single machine. However, the information must be available to all machines on the network even when the master machine was down. This basically restricted the distribution mechanism to mirrored disk files or NIS maps.

[1] It is easy to implement this by changing the master NIS file via cron.

Given the requirements the only solution possible seemed to revolve around extensive use of NIS, NFS and automount. Since we were already using these tools and had reasonable experience with them, this solution held no fears.

Setting up automounted NFS home directories and a shared mail directory was pretty straightforward, although care must be taken with mail to ensure network wide file locking works correctly. We solved the mailbox locking problem by ensuring all mail delivery occurred via a locally written "binmail" and restricting users to a single user agent, "elm". Both "binmail" and "elm" use advisory locking via the "lockf" function which allows locking of NFS files.

4 Basic tools

We decided to build two NIS maps, one to control machine access and another to control resource limits. The control files resided on the NIS master and are processed into NIS maps every hour if required.

All access checks and resource limit manipulation is done in a small program, called "fsh" (fake shell), designed to function as a login shell. All users (apart from root) have their login shell set to "fsh". After "fsh" has finished setting up the users resource limits it exec's "/bin/sh". Users are not permitted to change their login shell. We provide a mechanism for users to end up in their desired shell by setting parameters in their profile.

The approach of setting up a front end shell such as "fsh" as a login shell has several advantages from the system administration viewpoint. For example, since a localized path is set in fsh, users of "rsh" find that they no longer have to specify full pathnames to get local executables. Also resource limits get set even on processes started via "rsh" which would not be true if the code were incorporated into "login".

5 Implementation

5.1 NIS maps

The first of the NIS needed was the access control map. The map is keyed by the full hostname and thus contains one entry for each machine in the network. The basic structure of the map is a set of strings of the form *parameter=value* separated by whitespace. Case is ignored in map text, NIS map keys are case sensitive.

The **type** field is used to classify machines into various classes. If the **type** field is **personal** the machine is a personal (non-public) workstation usually located in someone's office. In this case access is restricted to those users listed in the **users** parameter. For example,

```
draci.cs.uow.edu.au     type=personal users=pdg,jag
```

If the **type** field is not **personal** the parameters used to control access to the machine are taken from the set **deny_group**, **allow_group**, **allow_user** and **deny_user**.

The value of the above parameters are regular expressions which are matched against the users group name list and user name. The list is scanned left to right to completion and the final status of an allow/deny flag is used to determine if access is permitted.

The following example illustrates the way the map works.

```
#
# First deny everybody then allow csci accounts
#
wraith.cs.uow.edu.au    type=server deny_group=.* allow_group=csci.*
#
# Allow all except engineering students
#
wyvern.cs.uow.edu.au    type=workstation allow_group=.* deny_group=eng.*
#
# Allow all csci students except David Bowie (dbowie)
#
bat.cs.uow.edu.au   type=server deny_group=.* allow_group=csci.* deny_user=dbowie
```

There can be any number (up the the limit of NIS) of access control parameters in the map. Each type of parameter may occur any number of times.

The second NIS map is used to implement resource control. The key for the resource map consists of two parts. The first part is a value of machine class as specified on the access control map or null. The second part is either a user name, a group name or "default".

The map itself consists of *name=value* pairs where name is one of **cpu**, **fsize**, **data**, **stack**, **core**, **rss**, **nofile** or **disk**. All but the last of the above names correspond directly to the BSD resource limits with the same name, scaled if necessary into convenient units. The last refers to a disk quota, in Megabytes.

The map is accessed by the following keys, in the following order:

```
user-name
group                           (loop over group list)
machine-class.user-name
machine-class.group             (loop over group list)
machine-class.default
default
```

In each case where the NIS lookup is successful, the line is scanned for values of resource limits not already set. Thus if there is an entry in the map for a particular user, (keyed by user name) the limits specified override those found on map entries later in the above sequence. The following fragment of resource map demonstrates the idea.

```
#
# user john has a large disk quota but otherwise gets the staff limits
#
john    disk=60
#
# mary wants large cpu limits but only on workstations
#
workstation.mary        cpu=600
#
# CSCI Staff all get reasonable limits on servers
#
server.csci     cpu=30 disk=50
#
# but larger cpu limit on workstaions
#
workstation.csci        cpu=120
#
# Students get the same limits no matter what
#
csci-stud       cpu=10 disk=5
#
# Everybody gets the same for stack etc
#
default         fsize=5000 data=5000 stack=5000 core=0 rss=2000 nofile=64
#
```

Disk quotas are handled by a small program which runs nightly on all diskfull machines as part of a cleanup script. For all users whose home directory is on the machine in question it checks to see if the disk quota they have matches the quota specified in the resource map and if not, sets the quota. The main requirement here was we did not want to have to remember to set up a disk quota for all new accounts. The system can not handle disk quotas on anything other than home directories.

All the code to handle the NIS maps (as well as many "useful" utilities) is incorporated into a shared library. Thus many other locally written or modified utilities can easily extract desired information from the NIS maps. The shared library also contains routines to determine if the user is an undergraduate, postgraduate, staff or guest. All programs which alter their behavior depending on the user call the shared library to determine the users class. In this way, we can alter the way the password file is organized and only have to update the code in the shared library for the new scheme to work correctly.

Jobs started via "cron" bypass "fsh" and thus provide a convenient back door to allow users to run processes that require more CPU time than the users quota allows. Since access to "cron" is controlled this presents no real problem in terms of security.

6 Conclusion

The system has worked surprisingly well. It is extremely convenient to be able to control access to all machines from a central location easily.

Because all users have "fsh" as a login shell it can be used to control other aspects of the login procedure. For example, we reserve our machines on friday mornings for scheduled maintainance. "fsh" refuses login to all but the system administrators at those times the machine is reserved with an informative message detailing when the machine will become available.

Although only in its first few weeks of operation, the scheme appears to be fulfilling its tasks of reducing the time required for system administration.

UNIX in the 21st Century

John Lions

University of New South Wales, Kensington NSW 2033

ABSTRACT

Long-term developments in the UNIX system will be influenced by technological changes in computer architecture and in communications. Several possibilities are speculated on.

1. Introduction

It is roughly 23 years since UNIX was born. What will happen to it in the next 23 years? Will UNIX exist 23 years from now? If so, how will UNIX change between now and then?

In racing parlance, UNIX is a horse that has come from nowhere to become the front-runner in the Open System Stakes. In spite of a sturdy sire (Multics), for its first ten years UNIX was a rank outsider, an apparent footnote to history, a trademark, admired in universities and some small enterprises, but not highly regarded in the world at large. By 1989, at the age of twenty, UNIX had become front-runner in the Operating System Stakes. Will the 1999 Open System Stakes be the last race that ever needs to be run?

Anyone who would answer 'yes' to that question would be brave indeed. The computer field is developing even more rapidly than ever, and much will happen in the next 23 years. In the year 2015, A.T. & T. will undoubtedly still exist and will still be interested in making a buck using its most valuable trademark.

Does UNIX have to change to survive? The answer is 'yes', because part of UNIX's survival strategy is that, having achieved fame and fortune as the world's first deservedly portable operating system, it should remain the software system of choice for companies that want primarily to build (the latest and most advanced) hardware. UNIX's evolution will be closely linked to developments in computing hardware.

UNIX's evolution will also be closely linked to developments in communications. It is perhaps no accident that A.T. & T. was a telephone company, not a computer company, when UNIX was born. During the last 23 years, the telephone network has been transformed and enlarged into a powerful communications network. Soon the amount of data that will be slopping about the world's networks will be incredible by the standards of just a few years ago. Much of it will be the equivalent of Marie Antoinette's cake: entertainment for the masses, but much of it will be a treasure trove for those who can tap into it, and absorb it.

2. Computing Hardware

Microprocessor technology has now reached the stage where the fastest and newest CPUs can be fabricated on a single chip. The 32 bit processor is giving way to the 64 bit processor. Soon it will be possible to fabricate many different processors on a single chip, and to surround them by arrays 64 Mbit memory chips. The power will be awesome. Will we need all that power, and if so, what for?

Computer architecture is about to enter an interesting phase: I predict that one important area that is about to change, and hence is ripe for innovation, is the way information is represented digitally. The encoding of numbers in binary form was not always as 'obvious' or as 'natural' as

it seems today. Few people seem to recall that 'bit' is a contraction of 'binary digit'.

- ASCII was a 1960's reaction to the ad hoc development of alphabetic character sets such as IBM's BCD. Today 95 graphic characters are not early enough. Japanese uses 10,000 characters or so. Sixteen bit characters are a serious possibility ... but is anyone seriously considering a 16 bit byte? If the byte size is to change, what should it change to? I see two serious possibilities: one bit, or 64 bits.
- One bit bytes would allow the whole of memory to be addressable as bit strings. However most useful data would have to be stored as aggregations of bits. Extended precision arithmetic would presumably be easy. On the other hand, CPU performance might suffer.
- The 64 bit byte has several attractions: it is large enough that most useful data items can be stored within a single byte: integers, addresses, instructions, characters, floating-point numbers. Only the last might be considered doubtful: some numeric applications seem to need extended precision beyond 64 bits. Most integers can be stored comfortably within 32 bits. 64 bits instructions have room for say 48 bit virtual addresses which seems to be enough for a few years (even 23 years?). Storing character data might seem to be the most doubtful of the possibilities. Moving beyond the ASCII character set now seems to be climbing onto some agendas. Seven bits are too few, but has anyone proposed an attractive alternative to the typewriter keyboard yet? Until they do, even 150 distinct graphic characters may be too many! Character data beyond 16 bits might be 'padded' by font and size parameters ...
- Compressing files before storing or transmission has many attractions. One early scheme was Huffman coding in which eight bit bytes are mapped into variable length bit strings. Later work suggested that other methods (Lempel-Ziv, or arithmetic coding) could be more effective. Even more recent work suggests that Huffman coding applied to (say) 16 bit bytes can be just as effective, or even better than these.
- Conventional communications can serve existing applications, e.g. finance and accounting, or telephony, very well, but some applications such as video transmissions are difficult to compress into a reasonable bandwidth. HDTV signals are of course even harder. This challenge can be met by new and revised methods for the encoding of information in digital form.
- Access to data in distant locations will certainly be possible, but can the data be trusted? Encryption schemes exist and many companies already use them, but there are known and suspected methods of attacking the standard encryption methods. If the CIA knows something about DES that we don't, other coding techniques are needed. How does one prevent accidental disclosure of information? Just using eleven bit characters that do not mesh with the ASCII character set would be a start. Compressing the data is also useful way of hiding it from the casual observer.
- PCM is an encoding method for telephony signals: 8 bit samples, 8000 times per second. A.T. & T. can now do much better than the 64,000 bps channels that it initially proposed for trunk telephony; it wouldn't make the same mistake again today (selling 64,000 bps as a single voice channel when it could be selling 16,000 bps or less).

3. Communications

The advent of high band-width communication channels on a global scale will allow computers to draw data rapidly from widely separated sources. Once the Telcos solve the problem of

marketing their newly expanded resources in an orderly manner, the location of the primary sources for many varied and useful data files may become immaterial.

However the provision of secondary sources that provide copies of the primary sources will be an important problem. Many of the problems in managing multiple secondary copies have been tackled already by the designers of hardware cache memories. Tackling the problem in software via networks that can be unreliable at times, and incur delays, is different.

The UNIX file abstraction has two aspects: a *content* that is an ordered string of bytes, and a set of *names*. In UNIX, files usually have one name, but additional names can be generated using the *link* system call. Files with no names (resulting from the use of *unlink*) may exist fleetingly but need not concern us here.

Fifteen years ago, if a file was created by copying another file, then one or other of the files was intended to be changed very soon (else why not use *link*). Hence there was no use in recording the source with the copy. Today the situation can be different: the copy may have been made using network resources, that were expensive in time, and so, once these resources have been spent we would like to keep the copy near-at-hand, at least for a while. Later we may decide that the local copy is superfluous. At that time we decide to re-use the space but to keep the name. This second case is familiar as it establishes a *symbolic link*, i.e. an entry in a local directory that points to another file. But what of the situation before that? Then we had both a local file and the name of a distant one ...

Suppose that before resuming the space occupied by our local copy we decide to confirm that (a) the original file still exists; and if so, that (b) it has not been changed in the meantime. If 'yes', we resume the space, but remember the name and the date. If 'no', we retain the space and change the status of the file from *copy* to *original*. Do we record where the file came from originally? (We may one day discover that there are many copies around.) Should all files have a pedigree? Can access to the pedigree be restricted more than access to the file itself? If the pedigree is itself a string of bytes, how will this string be associated with the string that represents the content? ... the previous content?

Suppose we keep the file, but choose to change its representation, e.g. by compressing it. What is its status now? Suppose we decide also to encrypt the file. What is its status then?

Many files have been copied many times (just think of your favourite editor). It is inconceivable that all copies of such programs will disappear before the end of the millenium. Suppose such programs were declared to be *itinerant* and could wander freely through the network. If you currently don't have one, but need one, wait for the next copy to wander by, or else broadcast a request to your neighbours, who in turn may borrow from their neighbours, etc.

4. Conclusion

I hope I have been able to suggest ways in which operating systems in general, UNIX in particular, may evolve in the future. Current developments are leading, so I believe, to a world where plain old ASCII files and communications will no longer be the norm. If a green, ASCII-free UNIX is the way of the future, will we still recognise it in 2015?

---

auug.sum.92 3—18—1992

Bruced — remote, reliable system administration

R. Loyzaga

Basser Department of Computer Science, Sydney University

1. Introduction

Bruced is a collection of *RPC* based daemons and utility programs that provide a network-wide administration system. The basic goal of this system is to facilitate the administrator in tasks that need to be replicated across the range of computers available on the network. These tasks have been difficult to accomplish because current methods rely on the availability of all target hosts to maintain synchrony. The fast rate of change and addition to the department's computing resources has also complicated many administrative tasks.

2. Motivation

The Basser Department of Computer Science currently supports a user population of over 1000, on 17 CPU servers with 3 different architectures (MIPS, SPARC, SUN3) plus a group of 20 workstations. Some of these servers are grouped into ''co-operative systems'' which are used by a specific group of users, i.e. Computer Science 1. This grouping requires the propagation of password changes and home directories, to all co-operating servers.

Administrating these systems has over the years lead to ad hoc solutions to the many mundane administrative tasks. The many additions and deletions to the set of departmental servers quickly led to the situation where some hosts were out of date or misconfigured with no easy way of maintaining consistency across a set of systems. The many workstations, and the fact that some users enjoyed turning them off, aggravated this situation. Clearly some way of automating the many administrative tasks was required.

3. Available Solutions

Many systems have been devised to help administrators cope with the many tasks required of them in a distributed environment. These systems however either address a small part of the distributed administration problem (i.e. passwd files), or require a great deal of system knowledge to be compiled into the software.

3.1 Yellow Pages/NIS

NIS is a popular solution to the distributed data file problem and is available on most workstations and fileservers. It is used to distribute files such as the **passwd** and **group** files across a range of participating hosts.

NIS[1] uses a master server which holds the actual data files, this server is responsible for propagating this information to client systems. Unfortunately, all the client systems require changes to the standard layout of these data files to participate in an NIS system. The standard libraries have been modified to interpret these data files and to connect to

the master server if necessary to obtain any information not held locally.

Some of the many drawbacks of NIS are:

i. All programs that deal with the ''distributed'' data files need to carry the NIS library ''baggage'' (around 60kb on a MIPS).

ii. Some innocent looking programs (**ls**) can be affected by the vagaries of network connections even when dealing with local files.

iii. Network traffic is increased to service information that should be known locally.

iv. Client ''maps'', which are local database images of the NIS files are sent infrequently to the client machines, and may be out of date.

v. The only service properly supported is password changing via a separate daemon (**yppasswdd**). Changes to other files need to be either propagated by hand, or during the infrequent scheduled updates.

To allow NIS to scale to more than a few dozen hosts, ''slave'' servers can be installed, which can serve a set of clients from a probably out of date set of NIS data files.

3.2 ACMAINT

ACMAINT[2] was developed at Purdue University to manage account creation and maintenance. It services a network of 25 CPU servers and 300 Sun workstations. The key features are a centralized account database which contains details on each user. A set of utility programs are provided to modify this database, the **passwd** program is modified to contact the central server process **DBD**, this central servers process then communicates to a per-host transaction daemon TRANSD which is responsible for making the required changes to the **/etc/passwd** and **/etc/group** files.

3.3 UDB

UDB[3] is a system based around a central database of user account information and has been made to work on a variety of operating system platforms. It was developed at Oklahoma State University to enable system-wide user account administration. It has the layout of the systems being maintained imbedded in to its configuration and it sends specific pieces of information to machines using application specific daemons on each host. UDB attempts to avoid the ''distributed'' database problem by centralizing all information and performing a set of known actions on each of the participating hosts based on this information.

Many other systems are implemented using this centralized database approach to reduce the administrative work required in setting up and deleting accounts, propagating new software and making department wide changes to computer systems, e.g. GAUD [4], newu[5].

4. The Bruced Approach

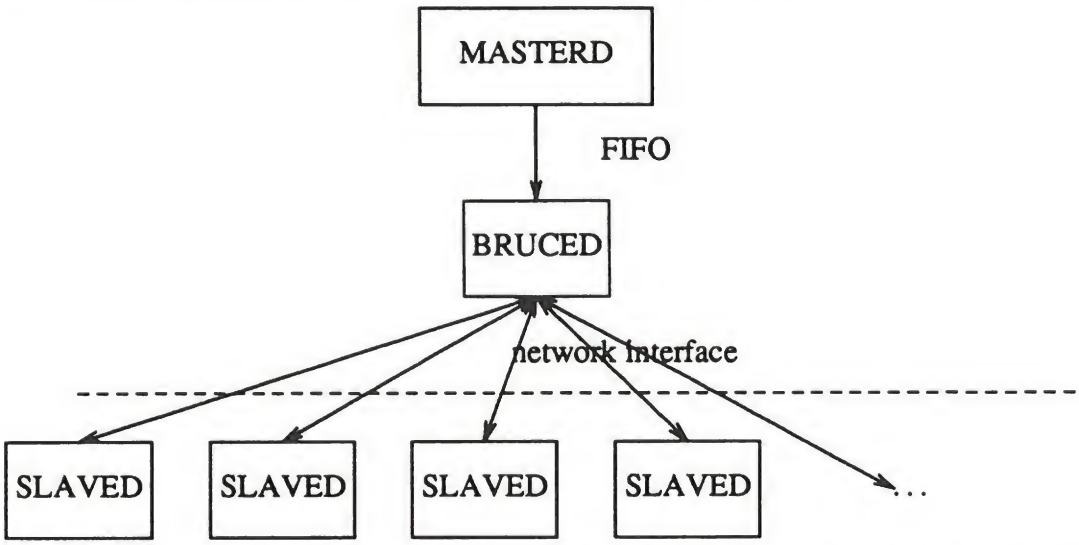
The basic approach taken in the development of **bruced** has been to avoid burying the complexity of the various system setups in the **bruced** software. This has been achieved

by only providing the mechanism for distributed control, not the administrative actions themselves.

The only facility that **bruced** provides is the ability to **reliably** distribute a set of command line arguments and a program **token** to the set of participating hosts. The actions taken by the hosts based on this **token** and arguments is purely a matter for the client machine.

5. The daemon hierarchy

The **Bruced** system is made up of three different daemons, two of which (**masterd, bruced**) only exist on a central network node (called the master host), and the other (**slaved**) exists on all participating hosts. (A recent change has been made to support ''relay'' hosts which should allow the system to scale up to hundreds of clients).



Each of these daemons performs a very simple task and as such the size of each of the daemons is of the order of 600 lines of C, much of which deals with the vagaries of RPC.

5.1 Masterd

Masterd has the sole job of creating "command" files for **Bruced** to process, these files exist for each target host. The files themselves contain data that will be used by the clients to provide a requested service. **Masterd** does not make use of any of this data, so these commands can be added to without changing the function of **masterd**. Client processes make use of a *RPC* call to **masterd** to start a new action. A simple interface to the *RPC* system is provided by the program **callmaster**, which simply calls **masterd** with its argument list.

After creating the per-host command files, **masterd** prompts **bruced** to indicate a new job is present via a FIFO and returns successfully to its *RPC* client.

5.2 Bruced

Bruced provides the job sequencing required to implement a reliable remote management system. Having received a prompt from **masterd**, **bruced** runs through the set of command files, attempting to connect to the **slaved** daemons on the target systems. These command files are removed once they have completed successfully.

Bruced will retry a command if it fails, it will back-off retrys up to a maximum of 120 seconds so as not to cause thrashing on a command that has no chance of success. Any host that fails to perform a command is eliminated from further processing of later commands until the command succeeds. This ensures that if the operations are atomic, no loss of newer data can result. **Bruced** attempts to contact several **slaved** processes at a time. This reduces the latency of command executions where some remote hosts are down.

5.3 Slaved

Slaved receives *RPC* requests from **bruced** and uses the first argument as a "key" to search a table (/etc/local/lib/slaved.tbl). The search, if successful, yields the name of a program to be executed. The remaining arguments are passed to this program.

6. An Example - password changing

To implement a set of grouped systems, each essentially sharing passwd file information, two programs need to be provided. The first is a replacement of the current *passwd* program, which asks the user for a new password after performing some checks. This program has been changed so that instead of modifying the */etc/passwd* file directly, it instead invokes **callmaster** with the arguments of the user name and the new encrypted string.

If successful, this command ensures that every machine will have a *RPC* call made to **slaved** with the arguments made available being "PASSWD host lname encrypted-passwd". The key (PASSWD) is stripped off before being passed onto the service utility. The file /etc/local/lib/slaved.tbl will probably contain a line such as:

```
PASSWD      /etc/local/bin/chpw
```

This means that *chpw* is invoked and it will do the update. In this case *chpw* is just a shell script that locks the passwd file, performs the required editing and unlocks the passwd file. Because the invoking host is available as the first argument, it can be used to test against to impose some "clustering" of password information.

A simplified version of *chpw* follows:

```
#!/bin/sh
case $1 in
mango|chico|zeppo|harpo)
        if lock basser /etc/passwd
        then
                :
        else
                exit 1
        fi
        if sed -e /'^'$2/s:'^'$2:['^':]*::$2:$3:: </etc/passwd >/tmp/passwd.$$
        then
                mv /tmp/passwd.$$ /etc/passwd
                unlock basser /etc/passwd
                exit 0
        else
                unlock basser /etc/passwd
                exit 1
        fi
        ;;
*)
        exit 0
        ;;
esac
```

The actual *chpw* is rather more complex as it must be able to cope with disk space exhaustion on both the root and tmp filesystems without a catastrophic failure.

7. Adding new utilities

New utilities that make use of the reliable execution facilities provided can be installed without changing any of the provided daemons. To add a new facility, just edit the *slaved.tbl* file to include a line of the form:

```
KEY  local pathname of executable
```

Then provide the front and back ends to this service. Once this has been done you can use the *reconfigmaster* command to signal to all the participating processes that they should re-read their configuration tables. This reconfiguration will also occur in the correct order, so that reconfiguration commands are not lost due to a machine outage.

8. Current Facilities

Administration utilities have been added to the **bruced** system once their repetitious nature has inspired the departments programmers. The current list of supported administrative actions are:

- Add a new user account on a set of systems;
- Propagate a user account to other systems;
- Change specific parts of a users passwd entry;
- Delete a user from a set of hosts;

- Execute a command on a set of hosts;
- Set up a mail alias on a set of hosts;
- Allow interactive editing of a merged set of passwd files and perform any necessary changes across all hosts;
- Allow system specific passwd file changes (i.e to a group of machines);
- Install a file to a set of hosts.

9. Debugging

All of the daemons make use of the facilities provided by **syslog** and produce messages of different **syslog** priorities. The most severe are at the "ALERT" level and are always logged on /usr/adm/messages. Other messages on the "INFO" and "DEBUG" levels may be logged onto a file to trace the activities of the various daemons.

Bruced and **masterd** create ".pid" files in /etc/local/lib/masterd when they are invoked, these are used to disable the daemons. They also change directories to /etc/local/lib/masterd, so that is the place for any core dumps that may occur.

10. Bruced status

Bruced has been in use now for over 2 years, in that time it has been responsible for the syncronisation of the departments distributed system data and for the distribution of software updates across the department's many systems. In that time is has served to remove some of the more mundane administrator tasks. It's best feature is the knowledge that once a broken server is repaired and reconnected to the network (even after many days), all the changes that should have occurred in that time will be propagated, in the correct order, probably before the login prompt appears.

[1] Sun Microsystems, *System and Network Administration*, May, 1988, pp. 349-387.

[2] David A. Curry, Samuel D. Kimery, Kent C. De La Croix, Jeffrey R. Schwab, ACMAINT: An Account Creation and Maintenance System for Distributed UNIX systems. *Usenix Conference Proceedings*, LISA IV, October 1990, Colorado Springs, pp. 1-9.

[3] Roland J. Stolfa, Mark J. Vasoll, UDB – User Data Base System, *Usenix Conference Proceedings*, LISA IV, October 1990, Colorado Springs, pp. 11-15.

[4] Michael Urban, GAUD: RAND's Group and User Database, *Usenix Conference Proceedings*, LISA IV, October 1990, Colorado Springs, pp. 17-22.

[5] Stephen P. Schaefer, Satyanarayana R. Vemulakonda, newu: Multi-host User Setup, *Usenix Conference Proceedings*, LISA IV, October 1990, Colorado Springs, pp. 17-22.

OpenEyes
A Performances Monitoring Tool for UNIX-based Systems

Perry Fisch, Masayuki Hatanaka,
Andy Law, Max Mattini
Unix Commercialisation
Group (UniComm)
Fujitsu Australia
tel: 61-2-410 4556

Abstract

The scalability of Unix from the PC level to the mainframe level is certainly one of the reasons which makes UNIX popular in Open Systems installations. However, the scalability of the operating system should be accompanied with a similar level of scalability in functionality. For example, the system administration/management functions of a UNIX mainframe should be a superset of the management functions of a UNIX workstation.

This paper addresses the performance monitoring aspect of systems management. It is based on the experience gained by the UniComm group during the development of an advanced prototype of OpenEyes, a performance monitoring tool.

1. Introduction

Performance reflects the amount of work completed per unit of time. Improving the performance of a system involves getting more work accomplished in the same time through the more effective utilisation of the available resources. In order to improve the overall performance (e.g tunning) of a system, first we should understand the systems themselves. By monitoring performance, OpenEyes helps a user gain in the understanding of the system. Monitoring is the first step toward automatic tunning, and expert system based, capacity planning in the future.

In Open Systems, computers of different sizes, vendors and missions co-exist 'glued' by the network. Preferably, the commercial users want to see 'The Computer System' supporting their business in a transparent manner. The users do not want to be aware of all the multitudes of hardware and software components which comprise their system. This transparent vision of the system/network should also be reflected in all aspects of system management.

In figure 1 (below), the operator controls the hosts on the network from a X-windows workstation. The performance data from the monitored systems is stored in a file server for a later analysis. Functionally, the multi-host administration is centralised, while physically, the software is distributed over the network.

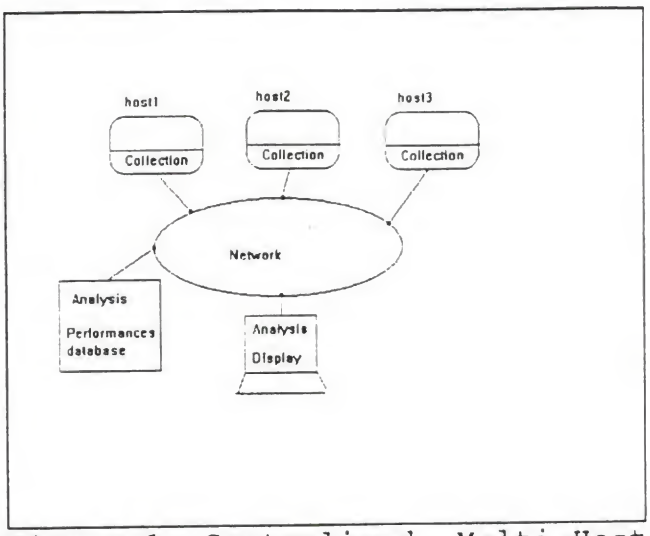


Figure 1 Centralised Multi-Host System Management

2. OpenEyes - Performance Monitor Tool

The OpenEyes prototype was used to monitor the performance and resource usage of Fujitsu's UTS/M or UXP/M hosts (UTS/M is based on UNIX V v2, while UXP is a mainframe implementation of UNIX System V v4.0). Up to 12 hosts can be monitored from one physical workstation. OpenEyes collects, analyses and records the relevant host performance information. It then displays this information to the user in a user friendly and graphical format.

Collector processes on the hosts (see figure 2) gather host information in an optimised fashion, while analyser processes on the workstation record, normalise and analyse the information. Finally, display processes on the workstation present the information using the OpenLook graphical user interface. The hosts and the OpenEyes workstation communicate via the TCP/IP protocols.

Two classes of monitoring capabilities are provided in OpenEyes: snapshots and trends. Snapshots provide a non-critical real-time 'picture' of the system. Trends show the evolution of performance across the time.

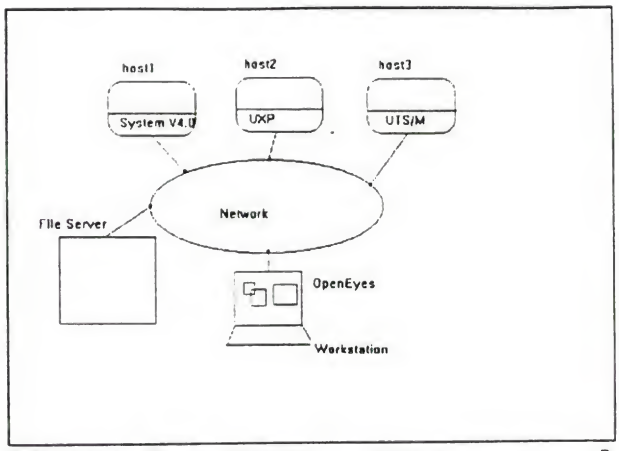


Figure 2 Open Eyes - A Centralised Performance Monitoring Tool

3. OpenEyes Framework

In order to support the concept of centralised multi-host management, OpenEyes is based on a modular framework (see figure 3 below). The framework isolates the hardware/system dependent aspects of performance and caters for emerging new hardware and operating systems. Tunning and capacity planning are not supported currently by OpenEyes but will be in the near future.

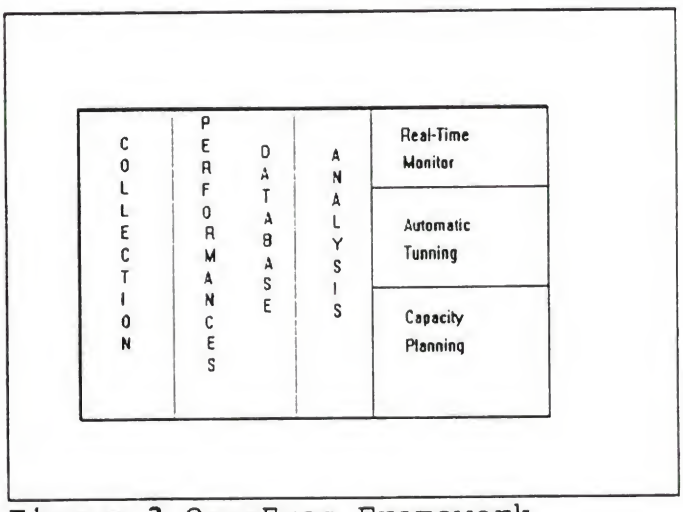


Figure 3 OpenEyes Framework

3.1 The Performance Database

"If it cannot be measured, it cannot be managed". The Performance Database is a collection of all performance and resource usage measurements, collected on all hosts during operation time. 'Operation time' means the time of operation of every monitored host in the system. The integrity and completeness of the Performance Database must be maintained in the event of a network failure.

The Performance Database is a logical database structured utilising the hierarchical concepts of subsystems, entities, and fields. The collection of related fields forms an entity, in turn, the collection of related entities forms a subsystem.

This logical database can be implemented using relational databases techniques or, later, using object oriented database techniques.

The structure of the Performance Database mirrors the major host subsystems:

- CPU subsystem
- Memory subsystem
- I/O subsystem
- Application/workload subsystem

3.2 The Collection Module

The task of the Collection module is the gathering of the raw data from the System V kernel or the various UNIX subsystems (such as a transaction monitor). Special care is taken in writing the collection module software so that the performance of the monitored host and the network are not affected.

3.3 The Analysis Module

The task of the Analysis module is to check the data received from the Collection module, to normalise the data, to insert the data in the database and finally to pass the data to the Display module.

Normalisation of data means the conversion of hardware/system dependent performance information into a uniform and logical format. Hopefully, in the future, international standards could define a common set of performance entities global to all hosts/operating systems.

3.4 The Display Module

The Display module is the user interface module of OpenEyes. It displays the information gathered on the host and transmitted across the network. Internally, the Display module processes the information received from the host, in order to present it to the user in a clear and easily readable format.

4. OpenEyes Functionality

OpenEyes Provides:

- Centralised multi-host monitoring.
- A hierarchical view of performance.
- Snapshots and trends graphics.
- Threshold control.
- Customisation.

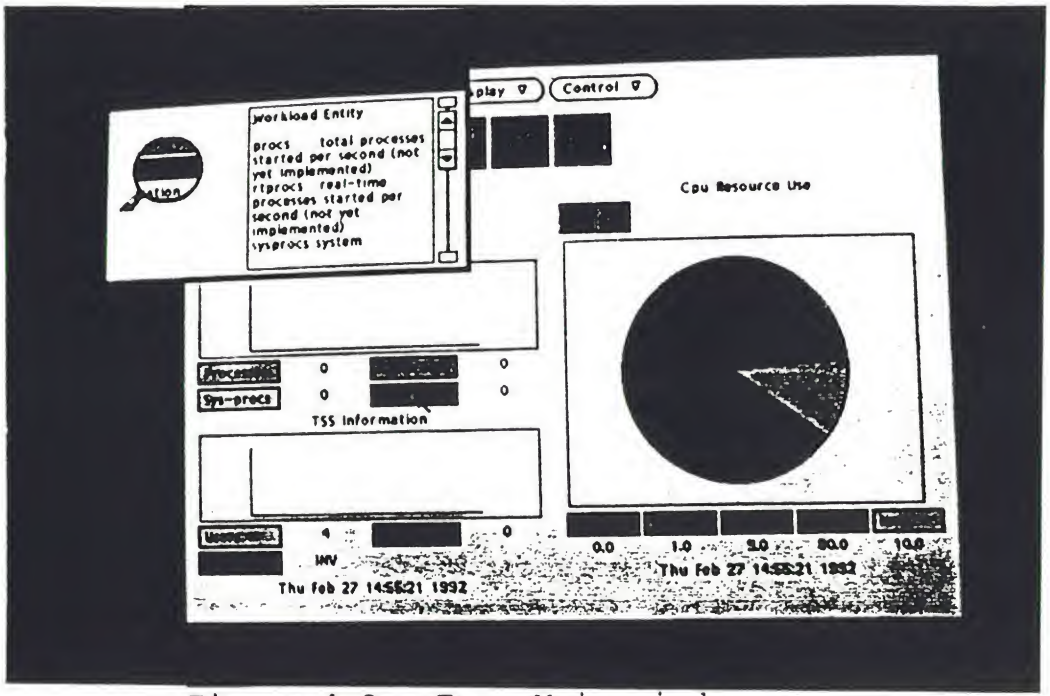


Figure 4 OpenEyes Main window

4.1 Hierarchical View of Performance

Centralised multi-host monitoring involves the management of a huge amount of information. In practice, one operator will monitor the performance of up to 12 hosts (minis, mainframes and workstations) on the network. For this reason the information should be presented to the operator in a structured fashion, separated by levels of abstraction, as needed, and in a user friendly X-Windows environment. For example, instead of having 6 control consoles to control 6 hosts, only one workstation is used. Instead of having the operator to poll the consoles looking for abnormal behaviour, the operator can define some threshold conditions and be notified when abnormalities happen. Instead of being lost in a 'flood' of numbers and data, a hierarchical and iconic interface is provided allowing the user to navigate throughout

the system.

Figure 5 (below) shows the file systems available on a given UXP mainframe. By selecting the file system icon, the operator can obtain the specific file system information.

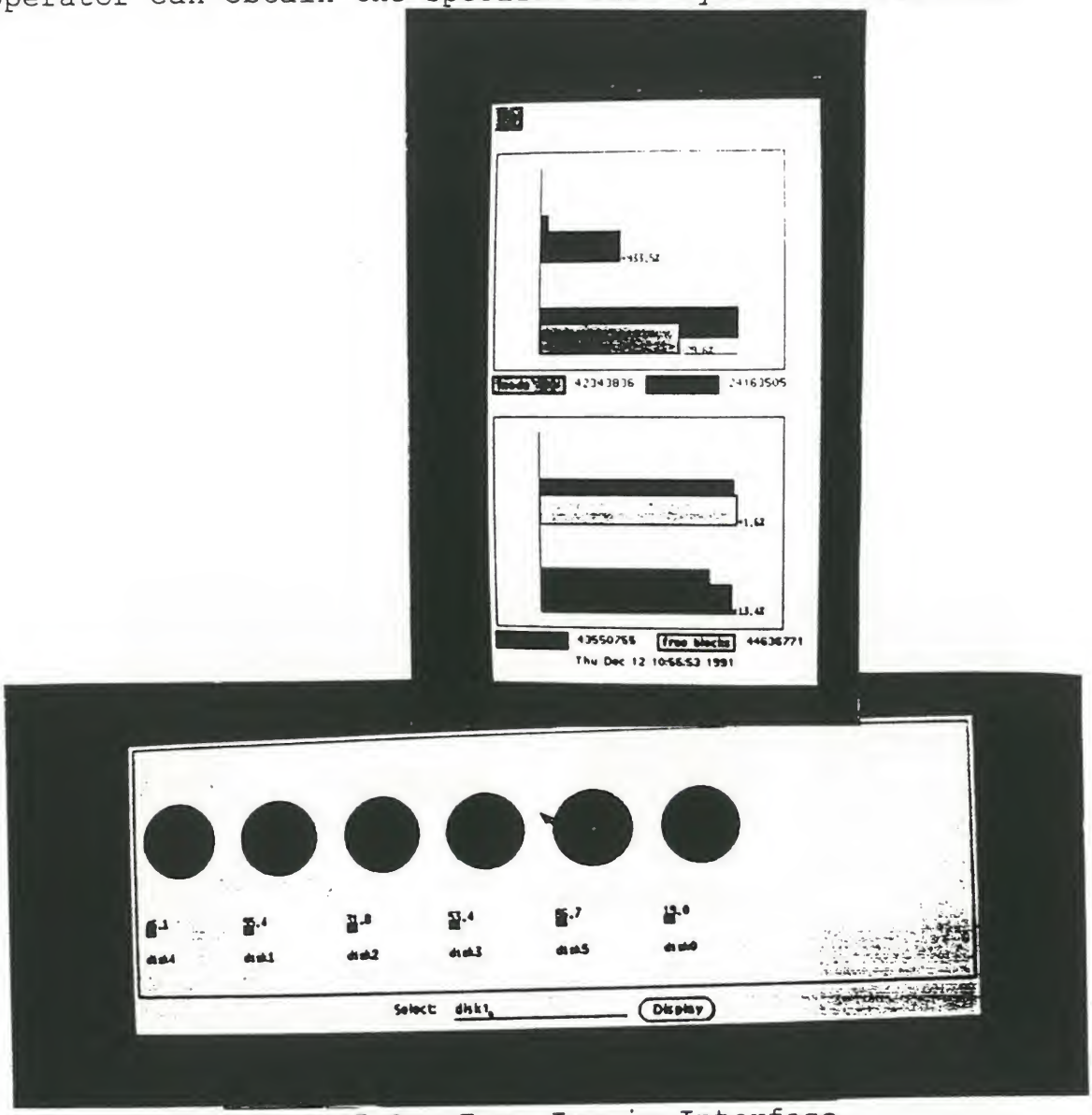


Figure 5 OpenEyes Iconic Interface for UXP File Systems

4.2 Snapshots and Trends

Figure 6 shows two classes of monitoring capabilities provided to the OpenEyes user: snapshots and trends. Snapshots provide a non-critical real-time view of the system. Trends show the evolution of performance and resources usage entities across the time.

Snapshots are of two types:

- Short period [5 second to 50 minutes]
  This type of snapshot determines the performance values during the last collection period (e.g the percent of CPU idle the last 10 seconds).

- Long period [1 hour to 24 hours]
  This type of snapshot gives the performance average during the long period (e.g the average percent of CPU idle for the last 12 hours).

By comparing the snapshots of the short period to the long period the user may detect abnormalities in the behaviour of the host. Both periods are user adjustable.

4.3 Threshold Control

The user can control the collection period of all of the entities. This is done simply by adjusting the sliders in the control window associated with each entity (see figure 7 below). In addition, the user can define the threshold values associated with every measured performance field.

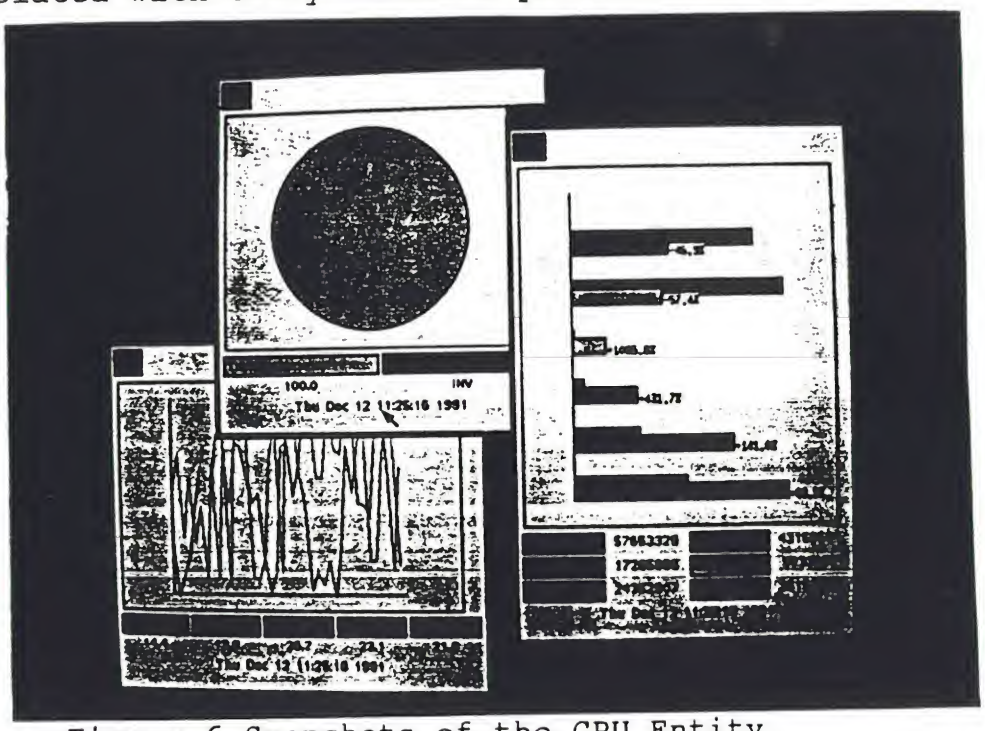


Figure 6 Snapshots of the CPU Entity

When the threshold value is violated the operator is notified of the event. An alarm/notification is triggered and is propagated. The alarm/notification mechanisms mirrors the logical database structure: the operator can detect/identify the origin of the alarm at the subsystem level, entity level or the field level.

5. Customisation and User Preferences

Customisation is of primary importance in monitoring a multi-host environment. This helps the operator associate the different graphical icons and windows with the different monitored hosts.

User customisation includes:

- The choice of entities to be monitored. For example, in a "panic" situation the operator may disable all monitored entities except the memory entity relieving the network from the added overhead.
- The style of graphical display (pie chart, histograms, bars) associated with each entity. Each kind of graphic presents the performance data in different way giving an added value to the raw data.
- The collection period associated with each subsystem.
- The threshold values associated with each field.
- Colour selections.

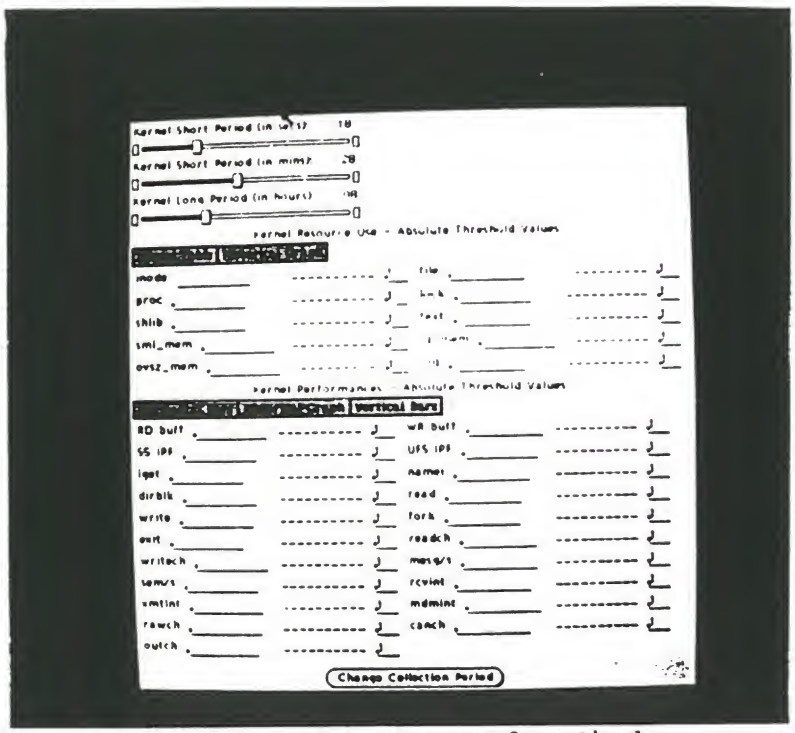


Figure 7 The Control Window: Collection Period and Threshold Values Definition

User preferences can be stored/retrieved across different monitoring sessions, relieving the operators from setting or resetting the environment every session and allowing them to tailor a default (standard) monitoring environment.

6. Conclusion

OpenEyes combines UNIX scalability, networking and the X-windows graphical user interface (GUI) to provide an elegant solution to centralised multi-host performance monitoring.

This iconic/graphical environment, presents the information to the operator in a 'easily absorbed' format making the understanding of system performance easier and user friendly. Finally, the measuring/understanding of system performance prepares the way for a more challenging task: automatic tunning and capacity planning.

Acknowledgments

OpenEyes is the first project of the Unicomm group in Fujitsu Australia. OpenEyes project's team members: Perry Fisch (Unicomm's Manager), Masayuki Hatanaka, Isao Hosokawa, Andy Law, Angus MacDonald and Max Mattini.

References

UNIX System V Release 4, System Administration's Guide
UNIX System V Release 4, System Administration's Reference Manual

Hardware Profiling of Kernel Network Code.

Andrew McRae

Megadata Pty Ltd.
andrew@megadata.mega.oz.au

This paper describes a method of accurately measuring and profiling kernel code (especially networking code) in real time. Some background is covered, which describes other more common (and easier) methods of profiling, and why these methods were rejected. Some goals are stated, and a proposed hardware/software solution is described. As an example, the profiling method is used to evaluate a kernel incorporating the Berkeley TCP/IP networking code; the final paper should contain the results of this exercise, showing how tracing of network software in real time highlights optimal or non-optimal code paths.

Warning to software people: this paper contains some descriptions of hardware.

Warning to non-kernel-hackers: this paper has lots of kernel hacking in it.

1. Optimisations.

Michael Jackson (the safe [non-dangerous] one) has said some things about optimisations.

Jackson's First Rule of Optimisation:

Don't do it.

Jackson's Second Rule of Optimisation (for very experienced programmers):

Think about it, then don't do it.

This expresses a well founded caution, often ignored by the naive, who would do well to remember the Prime Directive for Optimisation (as espoused by Kernighan and Plauger):

Make it right before you make it fast.

Even so, much effort goes into making programs as fast as possible, leading to a plethora of optimising pre-processors, compilers, assemblers etc. Given a poor (or slow) design, however, the best optimising compilers are not of great benefit. Experience has shown that if a piece of software is not performing, reviewing the design is often the best (only?) way of obtaining significant improvement.

Testing compiler optimisations is relatively easy; for a given set of test cases, the resultant instruction stream can be analysed to check for near-optimal code size or length of code path. Even given perfect translators, how can we test our designs in the same way? Generally programs are comprised of a number of algorithms, often interacting. How do we decide which algorithm is slow, or if a particular design section is far from optimal?

The key to all optimisation is to understand where or if it should be applied, and therefore the Golden Rule of Optimisation is:

Measure BEFORE you optimise.

UNIX† has a number of tools to help in this area; compiler profiling allows time based and function entry/exit profiling to be incorporated into programs, which then allow operating statistics to be extracted and analysed. Generally this is good enough for most programs, as the programs are not usually interacting with (or affected by) real world events. Simulators have also been used to effect by providing a much

† UNIX is a trademark of Bell Laboratories.

higher degree of granularity to profiling, allowing tracing of code paths etc. But often setting up and executing a simulation is difficult, or the simulator cannot provide for interaction with 'real time' events.

2. Kernel Measurement.

Kernels are a special case in that they must interface to certain real world entities, such as devices, networks, memories, clocks etc. Subtle and complex interactions occur between device drivers, processes and external events, and anyone who has attempted to fix bugs caused by these interactions will appreciate the difficulty in ascertaining hard data as to where optimisation is best applied. Kernel measurement has progressed to the point where it could be labelled A Black Art, instead of the realm of Wild Guesses. Kernels can be profiled, and through the use of statistics to count specific events, it is possible to do a fair job of deciding where a kernel is very slow, slow or not-as-slow-as-the-rest.

Some areas of kernels can be measured in the same way as user programs, using function counting and gross clock profiling, but these are not the interesting areas. What happens if one wishes to profile the clock interrupt code itself? What happens if you wish to measure the time taken to process character input interrupts, or discover the optimal code path taken for processing back-to-back packets through a certain protocol stack, checking the time to reply with acknowledgements?

The fly in the ointment is that kernel profiling is in the same vein as the Heisenberg Uncertainty Principle i.e the more accurate your measurements, the more you are perturbing the environment in which the kernel is running, and the less likely of getting data which reflects the actual state of the unprofiled kernel. Other methods are available which are non-intrusive, such as connecting large amounts of hardware to record the instruction stream; this is expensive and requires specialised hardware. Another problem with this method is that it often does not cope with cache effects; instruction caches must be turned off, thus ruining the non-intrusive nature of the measurement.

So we are faced with a dilemma; in order to rationally test kernel designs and code, we need accurate measurements, but in obtaining these measurements we change the environment of the kernel, and possibly introduce erroneous measurands (and consequently make wrong design decisions). Any kernel profiling system must be as non-intrusive as possible, or at least keep the effect of measurement to a minimum so that it does not grossly change the timing characteristics.

3. The Goals.

As a result of much software written in an embedded environment, a great deal of it driver and kernel related, I became increasingly interested in being able to easily measure and profile the software, and so make rational and informed judgements concerning algorithms and coding techniques. Faced with the regular need to discover why things were not responding at the expected speed, it quickly became clear that the human brain is not a good enough simulator to handle the complex timing interactions occurring within a kernel. Some early solutions to the problem was to use statistic counters, but this was usually too gross a measurement to help. Another favourite method was to press-gang a hardware design engineer to connect an oscilloscope to the equipment; this enabled finger-in-the-wind type measurements, and certainly helped when external hardware was being controlled.

Sophisticated tools such as logic analysers provided a major benefit, as whole sequences of events could be trapped and examined in the cold light of day. More intelligent software within the analysers also allowed instruction disassembly, which made easier work of following code paths, but this was generally tedious and unfriendly because of the difficulty in relating the raw instruction stream back to the source code. Other software can be made to perform time based profiling, but the sampling granularity was generally too coarse to be of any use.

In-circuit Emulators generally are considered the top of the heap for embedded development, and come with complete suites of cross-compilers, assemblers, remote debuggers and hardware which allows all manner of tracing and measuring programs. They also come with Rolls Royce price tags. Unfortunately they tend to be black boxes when it comes to analysing the data; it is often difficult to extract the desired information from the raw timing data.

I still had a need to find out what was happening inside our embedded computers, but I had a limited budget. By now I had tried several ways of getting the data, and I had formulated a wish list to describe

what I wanted.

- Fine granularity of measurement, so that accurate profiling may be obtained.
- Little or no intrusiveness, so that taking the measurement will not affect the timing of the kernel.
- Integration with development tools or program source so that source level code paths may be traced with ease.
- Profiling to occur for all kernel operations within a selected interval, including clock interrupts, device interrupts, even periods when processor interrupts were locked out.
- If some hardware assists were to be employed, then some easy and portable method of connection should be used e.g not having to connect 96 separate clips to a PCB.
- Immune to instruction cache effects. In fact it should still work as expected with instruction caching enabled (as any 'production' code would run the cache enabled).
- Granularity to a function level (however short the function is) should be the worst case; however it is desirable to also profile within functions if possible.
- Whilst the inital target should be a 68020 system, future systems employing other types of microprocessors should be capable of being profiled.

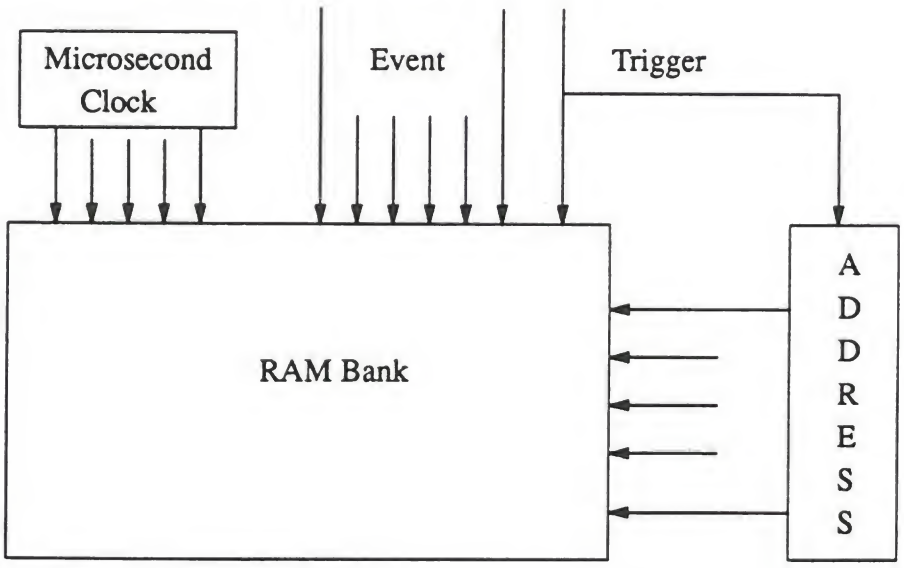
It became clear that it is impossible to fulfill these goals with software alone. It is also clear that complex hardware did not offer an elegant (or cheap) alternative This paper describes a solution to this problem which is a better alternative to software only kernel profiling, and much cheaper than specialised and complex ICE hardware measurements of kernel operation. It attempts to meet the above goals, and also be simple and cheap enough to build without much effort (even a software engineer could manage it).

4. The Profiler.

Three basic building blocks are used in the profiling system proposed; the first is a hardware device that is used to record events (stored in a RAM block). The second is a modified C compiler that allows event triggering code to be inserted into key locations, and finally the last building block is analysis software that is used to decode the backtrace of events and relate it to the source code.

4.1. The Hardware.

The role of the hardware in the Profiler is very simple. Its job is to store timing information and some identification value. A block diagram appears below.



When the identification code ('event') is presented to the Profiler, then it stores this code along with a microsecond clock value into RAM. The RAM address is automatically incremented every time an event is

stored, essentially storing the event/time in a large circular list. The list is 16K events long. The microsecond timer is 24 bits long, allowing a maximum time of 16 seconds between events before the time is wrapped around and information is lost. Note that this is the maximum time between events, not the total time that can be profiled - the analysis software only uses the timer value as an interval time, not as an absolute time. The event tag is 16 bits, allowing 65536 unique event tags.

The trick in this scheme is not the gathering or storing of the event/time data (a SMOH [Simple Matter Of Hardware]), but how to generate the event code, which must come from the equipment being measured. At first thought I had planned to detect specific instruction codes that represented function entry and exit points (e.g on the 680x0 every C function begins with a LINK statement, which translates to a hex instruction code of 4E56 - a return from subroutine translates to 4E75); this scheme had the advantage that is was totally non-instrusive but suffered from a number of shortcomings, such as inability to profile within subroutines.

It was clear that some software assist was required to actually trigger the profile events. One way was to place code at specific points to write the event value to a special location, but this assumes the equipment can actually perform the addressing required; the code translated to an instruction like:

```
MOVE.W    #val,EVENT_ADDR
```

All up this instruction cost 8 bytes, 4 read memory cycles and 1 write cycle (16 bit memory). It wasn't really a minimal trigger.

An elegant solution presented itself when I realised that the target equipment always has a boot EPROM located at location 0. Generally this was not used once the kernel relocated itself to RAM. It also presented itself as a simple method of connecting the Profiler to the equipment, just by piggy-backing a EPROM socket onto some cable, using the cable to bring the appropriate signals into the Profiler and then plugging the boot EPROM into the socket. The selecting of the boot EPROM is the event trigger, and the address presented is stored as the event data. Only 18 signal lines needed to be brought into the Profiler (16 address lines, the EPROM CE signal, and ground). Thus the event could be stored using the instruction:

```
TST.B    #event_val
```

Because `event_val` is accessing EPROM at location 0, it can use 16 bit absolute addressing. Total instruction length is 4 bytes, 2 16 bit reads, 1 8 bit read in total. On a 32 bit memory bus there is minimal overhead in using this event trigger.

Once the data is in the RAM, how do we get it out? In the initial instance, the RAM was plugged in using SmartSockets (i.e battery backup in the RAM sockets), so once the data was collected, the RAM was moved to another board and uploaded to a host. Later versions may connect a UART with some multiplexing, and upload via RS-232.

And so I had a workable hardware/software scheme that could record with accuracy specific events occurring, was easy to connect to a piece of equipment, didn't require lots of fiddly signal hooks, and the software trigger was minimal enough not to intrude very much in the timing of the kernel.

4.2. Generating the Triggers.

The next problem was how to manage the event triggers i.e how to automatically generate them in the target code, and how to manage the event value so that it could relate back to functions and points within functions. It seemed natural to place a trigger at the entry and exit of each function; that way code paths could be traced, and accumulated times calculated for each subroutine. It isn't really practicable to modify the source code to explicitly add the triggers; this would mean that a macro would have to be used so that the profiling could be turned off, and it would also mean manual allocation of a trigger value to each function, something that is tedious to manage. Besides, many functions have separate exit points, and often functions contain some initialization as part of their local variable declarations which would be performed before the trigger.

So it was decided to modify the compiler to add the trigger points; luckily the FSF's GNU C compiler was the compiler in use, and it was modified to generate triggers taken from a file containing the function names and values. Each function that appeared in the name file had a trigger generated at the start and end of the function. Another option to GCC placed a trigger in every function with the name and values

being written to a file. The name file contained flags indicating whether the value was a function entry, exit or an inline trigger. If extra profiling were needed within functions, then more triggers could be explicitly added.

4.3. Analysing the data.

Once the triggers were generated in the object code, and the Profiler captured some events, the raw data was then uploaded to a UNIX host. The data is processed by matching the event data (with the microsecond time values) with the function names. Several different types of reports can be generated e.g a complete code path trace can be shown, along with accumulated and separate function timings; a sample is shown below:

```
0:000 000   -> lanisr (280 us, 320 total)
0:000 080       ->  splnet (10 us)
0:000 200       ->  splx (10 us)
0:000 270       ->  schednetisr (20 us)
0:000 320   <-
0:000 325   -> softint (15 us, 1080 total)
0:000 335       ->  ipintr (300 us, 1065 total)
0:000 350           ->  splimp (10 us)
0:000 480           ->  incksum (25 us, 115 total)
0:000 500               ->  as_cksum (90 us)
0:000 595           <-
0:000 640           ->  tcp_input (...)
0:000 670               ->  incksum (20 us, 100 total)
0:000 680                   ->  as_cksum (80 us)
0:000 770               <-
0:000 805               ->  in_pcblookup (45 us)
          ...
0:001 400       <-
0:001 405   <-
```

The raw data can be massaged in many ways, such as graphing the average time of functions, measuring delays in networks, and how code paths are affected when multiple network packets are received.

5. Applications.

One of the first applications for the Profiler is to perform an analysis of an embedded kernel that Megadata has developed over the last few years. This kernel contains the Berkeley networking code to provide Ethernet and 802.4 Token Bus networking support, which is used as the basis for most UNIX kernel TCP/IP implementations. Through analysis of the code paths and timing information for a wide range of test cases, it is hoped that quantitative evidence will be gained to provide insight into the strength of the design decisions underlying the code, and also to guide any modification of the design or code. Since this protocol suite has been exhaustively tested and tuned, it is expected that little optimisation gains may be made. It will be interesting to see just where most of the time is spent, and to examine the various network layers and discover if improvements may be made.

The next (and hopefully much more fruitful) exercise is to apply the analysis to the OSI stack incorporated in the recent Second Berkeley Networking Release, which is the basis of BSD 4.4 (yet to be released). Due to the untuned nature of this code, and to the lack of long term experience with complete OSI implementations, it is expected that the Profiler will be a valuable tool in providing detailed information about how to optimise the code paths within the networking software.

Punya Palit

UNIX ON A FAULT TOLERANT PLATFORM

Abstract

This paper describes fault tolerant architecture of STRATUS and issues involved in running unix on this platform. STRATUS has been designed to support multiprocessors and duplexing of disk, memory and io subsystem. On the contrary, the traditional unix is centred around single processor architecture. The paper briefly describes the STRATUS architecture and then talks about the machine related issues like SMP (Symmetric Multi-Processing) Implementation, Writing Device Driver, Virtual Disk Layer, and Kernel Hardening. Some of the other fault tolerant utilities like maintenance daemon and rsn (remote service network) implemented under STRATUS are also mentioned.

Architecture

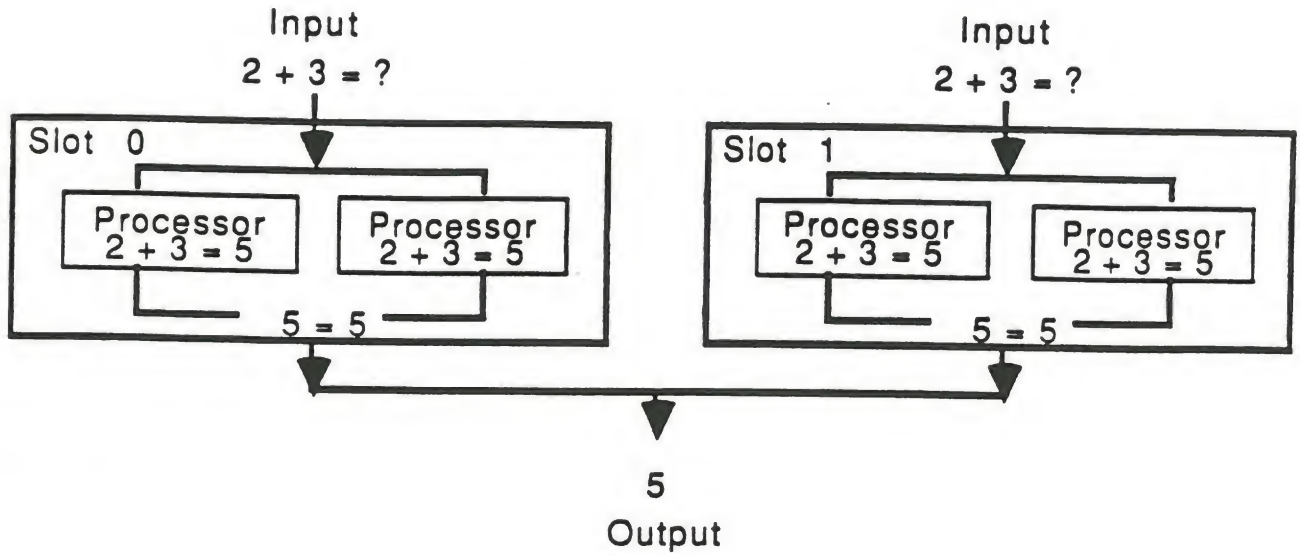
Stratus computers are based on multiprocessor design. There maybe (depending on the model) more than one CPU in the module, and the processing load is shared amongst all processors. Stratus achieves fault tolerance through the use of hardware. There are three distinct functions for hardware solution of continuous processing.

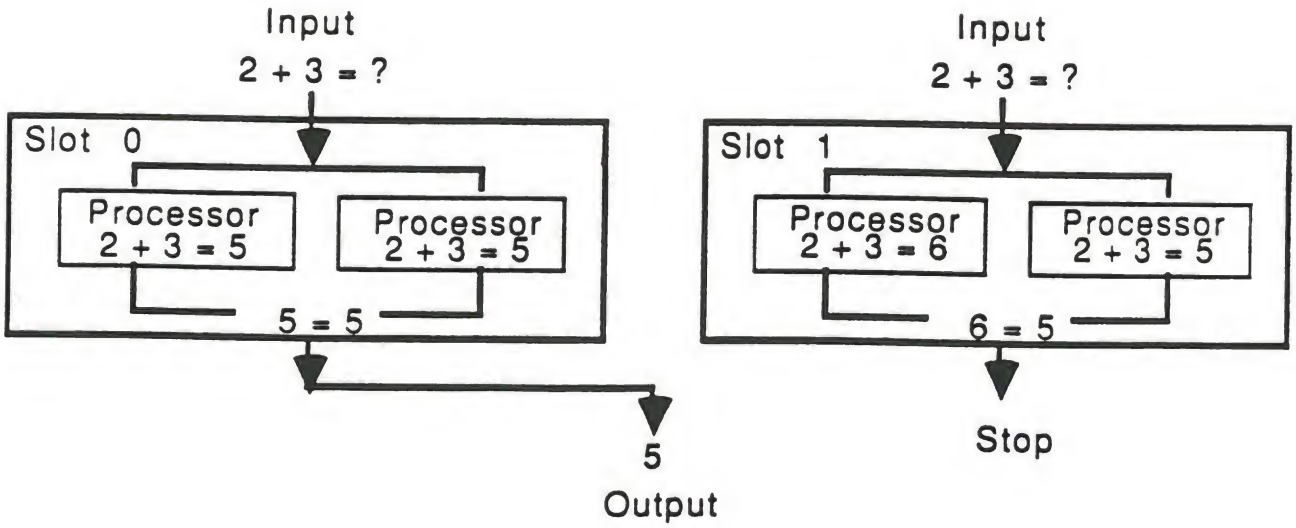
- Continuous Checking
- Duplexing Components
- Software Monitoring and Control

Main boards include two identical circuitary that perform every operation. The result of each set of parallel oprations is compared by the hardware. This checking process allows component malfunctions to be detected the instant they occur, so that the bad data never reaches the bus.

All major system components (CPUs, Memories, Controllers, and Disks) are duplexed. Each duplex component contains its own checking logic and diagnostics. When a board breaks its partner continues to run without any loss of data, performance, or software overhead. The Operating System identifies the broken board and runs diagnostics and if the board is bad, it is notified to the Customer Service Centre (CAC) through Remote Service Network (RSN).

The Partner Protects Against Single Failures





An extensive set of diagnostic routines are executed on each board every time the system is powered up. Maintenance software included in the Operating System receives all hardware maintenance interrupts, and then determines the cause and nature of malfunction. If the error is transient, the component is automatically restored to service. After the sixth transient error, the component is removed from service and an error message is sent to a selected terminal.

Unix on Stratus Platform

When Stratus decided to support unix on its fault tolerant platform and provide all the above mentioned features, there were two main issues to consider. Firstly unix kernel had to be modified to support multiprocessors and secondly any IO between peripherals and kernel had to go through IOP (IO Processor). RSN facility was implemented using uucp networking. rsnd is a special daemon process which checks the status of every hardware present in the machine by making a sysftx() system call to the kernel at a regular interval (10 secs). If a discripancy is noted, it immediately queues up a file transfer request to the CAC using uucp with necessary information. Stratus also has developed maintenance and dignostic daemon (MD) which provides fault tolerant services. MD is event driven user level daemon process which is responsible for management of maintenance state of the system. MD reacts to events like a) Administrative requests b) Insertions and removals of hardware components c) Hardware failures and errors and d) Power failure.

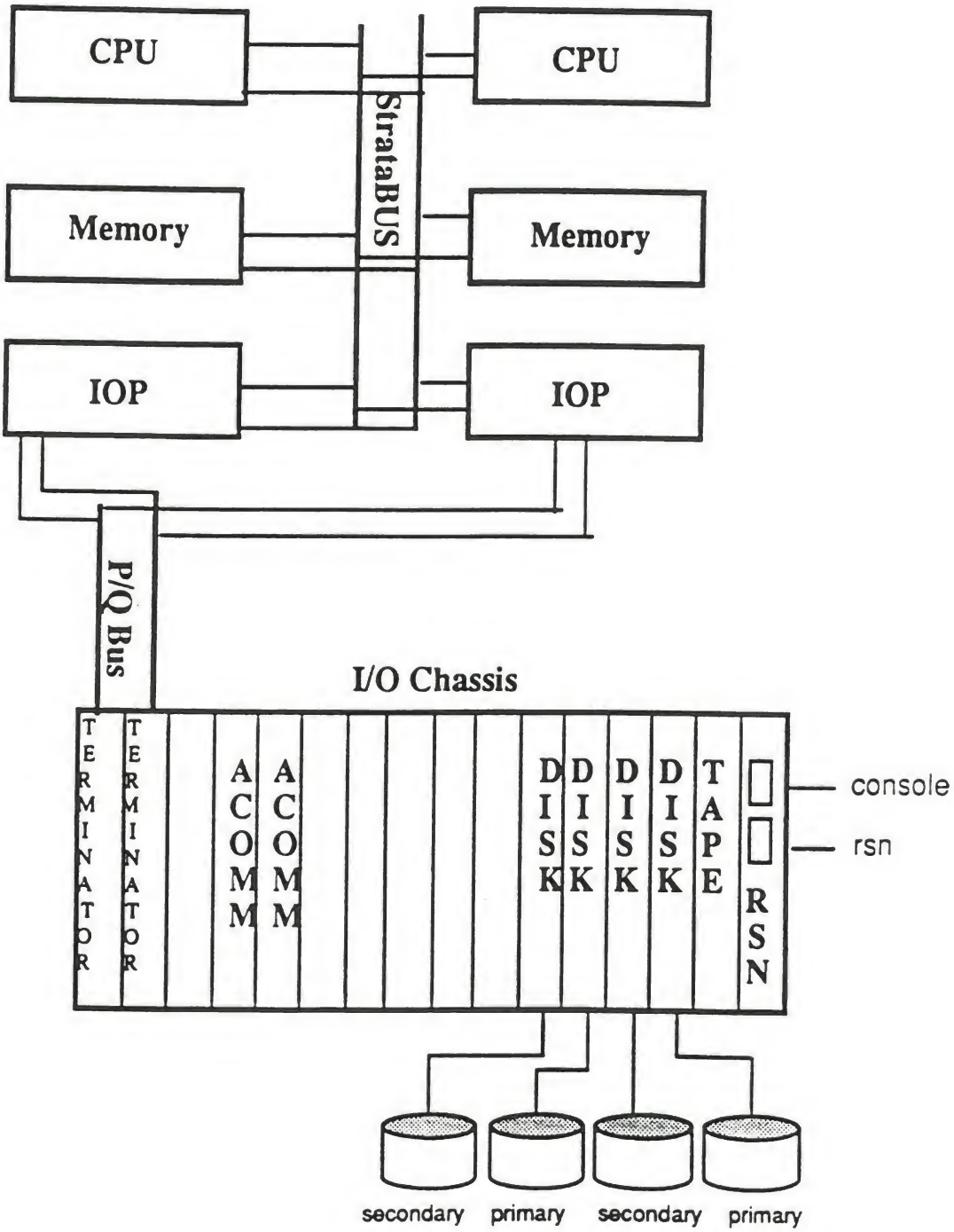
Virtual Disk Layer

It is a logical layer added between the file system and the disk subsystem and implement duplexing of disk without changing the user interface.

APPLICATION USER

File Subsystem
Virtual Disk Layer
Physical Disk Layer
Hardware

The Stratus Duplexed Architecture



User interface is unaffected by the virtual disk layer. User can also access the physical disk layer bypassing the virtual disk layer. VDL supports both simplex and duplex virtual disk. vdskconf utility is used to maintain virtual disk layer interface and FTX (Fault Tolerant Unix) kernel provides vdl routines to support this interface.

Writing Device Driver on FTX

When FTX is booted, it loads IOP firmware on IOP and the interface between kernel and IOP is provided by Device Access Layer (DAL). It is also necessary to know how to interface with Maintenance and Diagnostic Daemon (MD). MD is responsible to run diagnostic on the IOA card as well as downloading firmware. MD puts the IOA to ONLINE state before allowing device drivers to access the device. Because of IOP interface, it is not possible to add a foreign card to IOA chasis unless it is recognised by IOP firmware. Status provides special UCOMM cards for users to develop any special programs.

DAL is a set of kernel resident functions and designed to shield the device driver writer from IOP interface. It creates and maintains mailboxes for each device and each device is attached by the maintenance daemon. It provides unix device number (major/minor number) translation into slot and location numbers. It also provides an interrupt handler to service interrupts generated by IOP.

Symmetric Multiprocessing

Every available CPU can run the kernel as well as the user process. If the kernel can run concurrently on any of the available CPUs, it needs a protection scheme such that the data structures maintained by the kernel remains in a consistent state at all time. When a process goes into kernel state, it specifies the interrupt level and all the lower level interrupts are masked out and only the higher level interrupts are entertained. This strategy does not address protecting data structures if another interrupt is handled concurrently by another CPU. It also forbids another CPU to handle a lower priority interrupt. Three types of locks are provided on FTX to resolve this problem.

Hardware Locks : They are the lowest level locking protocol in FTX and is represented by 16 bit word. Reading a lock word is very expensive.

Exclusive Locks : The exclusive lock layer is built on top of the hardware lock layer. It can be acquired by at most one user or CPU. Subsequent users must wait until the lock is released.

Read/Write Locks : This allows single writer while allowing multiple readers.

Both types of locks can be locked/unlocked from callside and interrupt side. These locks are used to ensure data consistency of a global data structure or a critical section of a kernel code. Locks can be either spin or block type. In spinlock you spin until you get the lock. It should be used to acquire resources that you need for a short period of time. In block lock, you specify a spin count and if the resource is not available you go to sleep and wait for the other to release the lock.

Example

To increment global variable foo (Foo++), we do

```
SPINLOCK(foo_lock, SLEEPNOTOK);
foo++;
UNLOCK(foo_lock);
```

Locks must be placed between sleep/wakeup pair to gurantee atomicity. Illegal use of locks can cause system to panic.

Kernel Hardening

Kernel Hardening involves developing new software so that the kernel is isolated from any potential problems, and ensuring that the impact of any problems is minimized. Stratus has developed an extensive set of back-end processes so that hardware failures are simulated and this way performance of kernel is tested during a pcb failure. Stratus has also developed its own device drivers to isolate problems such that no device error on failure causes a kernel to panic.

SVR4.1 Enhanced Security: A Technical Overview

November 1991

Overview of UNIX(r) System V Release 4.1
Enhanced Security

Chris Schoettle
General Manager
UNIX System Laboratories
Australia & New Zealand

AT&T Australia
Level 23
60 Margaret Street
Sydney NSW 2000
Australia

Tel: +61-2-906-8953
Fax: +61-2-436-4673
Email: cts%uslp@usl.com

Contents

Introduction

The market for secure computer systems continues to be a fast paced and rapidly growing section of the industry. Specifications for secure systems come from all facets of the industry, governments both domestic and international, commercial enterprises, and standards organizations.

This paper describes the role that UNIX® System V Release 4.1 Enhanced Security (SVR4.1ES) plays in this marketplace. The main focus is on the feature changes made to UNIX SVR4.1ES to satisfy the Trusted Computer Security Evaluation Criteria (TCSEC) [1] while still maintaining the look and feel of traditional UNIX systems. A brief discussion of how SVR4.1ES meets major security specifications is also included.

Computer security has existed in some form or another for as long as sensitive, proprietary data has resided on computers. Before open systems, the majority of computers in the work place were stand-alone, mainframes running some version of the vendor's proprietary operating system. The security typically consisted of restricting system access by requiring some form of authentication, usually in the form of a user ID and perhaps an associated password. On some systems, data set access was also restricted by some form of authentication. Access to any system resource required a series of job control language (JCL) statements. The JCL itself was complex and varied between different systems; thus it served as a form of access restriction (since the naive user was unable to access data sets or system services). In this type of environment, the security afforded by identification and authentication was sufficient. Because these computers were running proprietary operating systems, the source code was unpublished and not well known. This lack of knowledge, little or no networking, and limited interoperability among the various proprietary operating systems made it difficult for rogue software to penetrate the operating system and spread to other systems.

The emergence of the UNIX system and open systems changed the way in which computers were used and configured. The UNIX systems tended to be small systems configured in some kind of networked cluster. By eliminating the need for complex, cumbersome JCL, most system services became accessible to even the most naive users. Since the systems were open, the

operating system source code was published and well known. Frequently, the system source code resided on the system and was left accessible to all users. These features changed the type and nature of the security threats. For example:

- Since the systems were open, system software could be obtained from sources other than the vendor, such as a university. The non-vendor software was not always placed through a rigorous design, development, test, and documentation cycle. As such, the non-vendor software sometimes contained holes through which rogue users could gain system access or system privilege.
- The networked environment, combined with the interoperability of the UNIX system, enabled the development of virus programs which could affect the entire network of machines. Additionally, users could use the network to evade detection by having the virus program executed remotely.
- Since the operating system source code was well known and easily available, it could be examined for holes through which access could be gained or privilege could be obtained. While holes existed on the proprietary systems, the lack of access to the source code prohibited thorough and extensive searches of the operating system.
- The ability to share data among users within a designated group or all the users on the system proved to be a very powerful, frequently abused feature. For example, files were commonly left accessible to all system users, when allowing access at the group level would have been sufficient. Since this type of abuse was also seen with system files (for example, **crontab** files left with a mode of 777), a new type of threat was introduced.
- Since no JCL was required to access system resources, users could frequently run the system out of space, out of processes, or tie up the system console by generating extraneous system messages. This enabled users to violate the system and prohibited the system administrator from taking preventive action.

New threats such as those described above required more sophisticated security mechanisms than the identification and authentication provided on the mainframes. Early UNIX systems solved these problems by extending the group mechanism (multiple groups), restricting of network access, and

adding resource limits. While sufficient for many applications, these were not sufficient for high security applications such as those used by the United States government. Seeing the need for assured system security, the United States government, specifically the Department of Defense, began to address the security issues related to networked, resource-sharing computer systems.

Interest in Computer Security

Interest in assured secure systems began within the United States government. As early as 1967, government recognized that resource-sharing computer systems brought a unique set of security problems that needed to be addressed. Growing out of efforts that largely came from Department of Defense (DoD) computer system needs, the National Computer Security Center (NCSC) was formed to identify problems and solutions for building, evaluating, and auditing secure computer systems. In 1983, the DoD published the definitive guideline to secure operating systems, the Trusted Computer System Evaluation Criteria (TCSEC), better known as the "Orange Book" because of its orange cover. The TCSEC contains the certification criteria that a system (software, hardware, and firmware) must meet for multilevel secure standards. A key requirement of the TCSEC at higher levels of trust is assured protection, that is, the system cannot merely claim to be secure, but must be proven to be so (via source code evaluation, covert channel analysis, penetration testing, and test suites). The assurance testing is performed by the NCSC against the security division claimed by the vendor submitting the system for evaluation.

The NCSC evaluates commercial operating systems against the TCSEC and assigns security ratings to them. The TCSEC defines four security divisions (D, C, B, and A—from the least to the most secure) containing numerical classes (for example, B1, B2, and B3—from least to most secure). The full range of ratings currently defined is, from least to most secure, D, C1, C2, B1, B2, B3, and A1. The NCSC separates function and assurance at each level; as the level of security becomes more rigid, the need for security increases.

Three U.S. government directives, National Security Decision Directive 145, Department of Defense Directive 5200.28, issued in 1988, and the National Telecommunications and Information Systems Security Committee's NTISSP no. 200, issued July 15, 1987, define C2 as the minimum level of protection required for a wide range of government computer systems. In addition, government procurements generally include a security requirement based on the TCSEC at the B1 or B2 level.

International governments, fearing the U.S. government would become the predominant force in computer security, began to define their own security criteria. By the late 1980's, the Canadian, British, French, Danish and German governments were actively defining computer security standards, with the German and British governments publishing criteria and performing evaluations of secure systems. In Britain, the Department of Trade and Industry published the Green Book. In Germany, the German Information Security Agency published the White Book, while the French developed the Blue-White-Red Book. The German and British governments have published criteria and perform evaluations against their own criteria. An outgrowth of the European security criteria work was the Information Technology Security Evaluation Criteria (ITSEC). The ITSEC was jointly developed by Germany, France, Britain, and the Netherlands and is being adopted by the European Economic Community (EEC) as the security criteria for a united Europe.

While the various criteria have a great deal in common, there are differences as well. For example, the German White Book, while similar to the TCSEC, has some requirements above and beyond the TCSEC. One of these is a requirement to support Access Control Lists (ACLs) beginning at the B1 level (a B3 TCSEC requirement). The German criteria also separates function and assurance.

Interest in computer security within the commercial and public sector resulted in the formation of special interest groups, standards organizations, and the definition of security requirements groups within the commercial sector. Banking groups within both the United States and Europe have defined requirements for more secure systems. A consortium headed by American Express and Electronic Data Corporation recently published the Commercial International Security Requirements (CISR). Consortiums of hardware vendors, such as Eurobit, have begun working toward the establishment of common security goals.

This high level of interest in secure systems led to the creation of several standards organizations. They were chartered to define interfaces and data definitions for secure systems. The International Standards Organization's Joint Technical Committee (ISO/JTC), IEEE/POSIX™, X/Open®, and /usr/group (now UniForum™) all commissioned working groups in these areas. In addition to the standards activity, the NCSC-sponsored Trusted UNIX Working Group (TRUSIX) activity spawned task forces to work on

Discretionary Access Control (DAC), auditing, and security policy model development.

In the user arena, American and international market interest in secure systems has grown immensely. At the recent ITSEC workshop, over 500 representatives of users, governments, commercial interests, vendors, universities, and the public sector were present. The one common interest expressed by all participants was the need for increased security in computer systems.

Government Security Needs

Through the TCSEC, government needs and the validation/certification process are well defined and understood. National Security Decision Directive 145 establishes the need within the government for secure systems meeting or exceeding the C2 level. Requests For Proposals (RFPs) in the last few years have been written so that C2/B1 levels would meet the requirements at the time of requisition, but B2 would be required in the future. The recent United States Navy and Air Force RFP specifies a broad range of features including auditing (C2), Mandatory Access Control (MAC), and High Assurance (B2).

Government agencies, like the Navy and Air Force, have requested security features beyond the C2/B1 level which provide auditing and minimal assurance. Most agencies are beginning to request systems with features at the B2 level, where additional security features provide penetration protection and increased data assurance.

Many recent RFPs, such as the recent Joint Army and Navy (JSAN) request, have begun to specify Compartmented Mode Workstations (CMW). Specifically, a CMW is a B1-level workstation with B1+ extensions. The CMW requirements go beyond the TCSEC in requiring a trusted window manager and a second, MAC-like label known as a CMW label (also commonly referred to as an information label).

Faced with the prospect of four different evaluation criteria in a united Europe, France, Britain, Denmark, and Germany recognized that this work needed to be approached jointly and that harmonized evaluation criteria should be defined. Because the basic concepts and approaches of the four countries were much the same, they decided to take the best features of work already done and place them in a single set of criteria, the ITSEC. The intent of ITSEC is to have a set of common criteria for use within at least the four countries involved and eventually the entire EEC. So, evaluations performed in Germany could be honored in England. The ITSEC's single set of security evaluation criteria will, in turn, yield to a single set of well defined functions which would be required to satisfy both American and EEC RFPs. Before this can happen, the United States and the other EEC countries must buy into the ITSEC. Based on the level of comments received on Draft 1 of the ITSEC at the recent ITSEC workshop, much work must be done on this document before it can be considered equal to the TCSEC. Draft 2 of the ITSEC is planned for mid-1991 while a companion document, the *ITSEC Evaluators Guide*, is planned although no formal date for release, has been announced.

Commercial Security Needs

Within the last few years, the commercial sector has begun to recognize the need for increased security in computer systems. They have recognized also that, while government and commercial requirements had much in common, they varied in some of their security needs. With this in mind, such areas as financial institutions, telecommunications, and service industries began to develop their own profiles of security needs. Bell Communications Research (Bellcore), on behalf of the regional telephone companies, has been very active in the security arena and has developed its own security profile--the Bellcore Standard Operating Environment (SOE). American Express, in conjunction with a consortium of 40 other companies, developed the Commercial International Security Requirements (CISR). Within the commercial market, two basic factors hold true. First, market needs will evolve to more secure systems (as they become available); second, the features required by the TCSEC at the B2 level represent a super set of the security features required for commercial use.

In most cases, the commercial sector has identified the need for increased computer security but has failed to map the need into the features required to

provide the security they want. One step toward defining these requirements has been the CISR.

Contact with commercial users, either via standards organizations or trade shows, indicates that, as more secure systems become available, the market will evolve to them. For example, American and European interest in UNIX System V/Multilevel Security (SV/MLS), AT&T Federal Systems' B1-evaluated product, has exceeded initial expectations.

An emerging sector in the late 1980's was the international market, specifically within the EEC. For example, the Commission of the European Communities recently published a set of requirements, the *Security Requirements for Extended POSIX Computers*, that defines a list of mandatory and optional requirements. These requirements must be present in a product before the EEC will consider purchasing it. Most of these requirements are equivalent to TCSEC B2 features.

Functionality vs Assurance

The major difference between the TCSEC and the ITSEC and between commercial and governmental specifications is the requirement for a high degree of assurance. The TCSEC and the United States government require that a system rating at the B level or above is secure functionally and provides high assurance (for example, extensive penetration testing and documentation).

After the B1 level, the requirements for assurance become far more stringent. At the B2 level, requirements for modularity, covert channel analysis, and limits on the number of trusted processes and global variables are introduced. The higher levels, B3 and A1, introduce even more stringent requirements, requiring formal proofs of correctness. The ITSEC, following the lead of the German security requirements, separates function and assurance into different rating criteria. Thus, for example, a vendor could submit a system which meets stringent functional requirements (F-B3, for example) but with a far lesser degree of assurance (E3, for example). The same vendor requesting an evaluation against the TCSEC would have to submit the system at the B1 level since it would not meet the B3 assurance criteria.

The non-TCSEC-mapped classes are non-hierarchical and are defined as F-IN (high integrity), F-AV (high availability), F-DI (safeguarding data during interchange), F-DC (confidentiality of data during interchange), and

F-DX (network data interchange). Effectiveness (or assurance) classes range from E0 (none) to E6 (high). Currently, SVR4.1ES maps to the F-B2/E4 ITSEC class.

Within the commercial sector, the need for the high degree of assurance associated with the higher levels of trust, specified by the TCSEC, has not yet been realized. While the need for the features has been identified, the need for the assurance that the features have been properly implemented has not. The commercial sector has not realized that the increased modularity provided to meet the increased assurance yields great benefits in portability and maintainability, and that the covert channel analysis serves to close holes in the system security policy. One must question how useful MAC labels are when covert channels exist where data can be easily reclassified.

The single biggest advantage B2 provides, when compared to other secure systems, is in the area of security assurance. Security assurance is, as the name implies, assurance that the system's security features work as advertised and are implemented properly. While all B levels require assurance, the level of assurance required at the B2 level is significantly greater than that required at the B1 level. At the B2 level, covert channel analysis, extensive penetration testing, and requirements on the number of global variables are introduced. Additionally, the system must meet stringent modularity requirements. While frequently dismissed as TCSEC requirements with little commercial use, these security assurances provide significant value. The penetration testing and covert channel analysis provide assurance that the system has no back doors by which the systems security policy could be circumvented. Obviously, a secure system is worthless if it contains back doors that allow users access to the system, allow undetected access of data, or allow the security policy to be circumvented. Reducing global variables makes it far less likely that a change in one module will introduce changes in other modules. The modularity requirements yield a system which is easier to maintain and port.

Covert Channel Analysis

A covert channel is any means through which the systems security policy can be circumvented by an unauthorized user. Unlike most "holes," undocumented covert channels are not audited; thus the exploitation almost always goes undetected. In high assurance systems, an extensive search for these channels, a covert channel analysis, is required. While many of the covert channels are obscure and hard to exploit, several are common, well known and exploitable

by the knowledgeable and persistent. A covert channel analysis identifies the channels such that the system vendor can eliminate the channel or make modifications to make the channel unusable. Thus by eliminating the methods by which the security policy can be circumvented the overall security of the system has been greatly increased.

The Role of Standards Organizations in Defining Computer Security

In addition to government agencies that define security criteria, a number of standards bodies are also involved in evaluation of security criteria. These standards organizations have chartered groups to define standards for data interchange, secure interface specifications, and evaluation criteria. ISO is currently working on specifications for the interchange of auditing data. IEEE POSIX has several groups working in the area of security. The primary group working on security, P1003.6, is defining application interfaces for auditing, MAC, CMW information labels, DAC, and least privilege. Other POSIX groups such as P1003.7 (system administration) and P1003.17 (network services) have spawned subcommittees and liaison activities to work with P1003.6 to resolve the issues related to security administration and secure networking.

X/Open's Security Working Group (SWG) recently published the *Security Interface Specifications: Auditing and Authentication* snapshot document. This document, which was co-authored by AT&T/UNIX System Laboratories (USL) and IBM, was distributed for industry review late last year and has been adopted by POSIX P1003.6 as the base for its auditing specification. TRUSIX has developed and published B3-level specifications and guidelines for ACLs and auditing. The TRUSIX auditing guidelines use much of the X/Open auditing work, while the TRUSIX ACL specification is based heavily on the SVR4.1ES ACL implementation.

SVR4.1 Enhanced Security Background

The UNIX system evolved in an open R&D environment where the free and easy exchange of information was paramount. Login accounts for guests without passwords, unprotected source and system files, and unrestricted data lines were typically found in these environments. From its inception, the UNIX system has always provided security features.

The original UNIX system featured Discretionary Access Control, which provided the user with the ability to control file access by setting file permission bits defining the access permission for the owner, group, and others. The system also provided an identification and authentication (I&A) scheme in which the system prompted the user for a system login identifier and an accompanying password.

Unfortunately, to facilitate unrestricted access, these features were frequently circumvented. Prime examples of this included programs with setuid/setgid bits, file modes left as 777, login scripts complete with login and password, and unrestricted guest logins. On systems where stringent security measures were in force for ease of execution and debugging, security was frequently compromised by setting applications to run with an effective UID of 0, thus making the application exempt from any system security requirements.

Since early UNIX systems featured little administrative support, the situation was made worse by administrative and operator errors which in turn, led to software holes through which hackers could gain unauthorized privileges. One example of this would be leaving the system's **crontab** writable by someone other than root. With the introduction of open networks came the ability to rapidly interconnect and freely share data among multiple systems. The combination of unprotected data files, holes by which privilege could be obtained, and easy and rapid interconnection, provided the ideal environment for virus attacks. Several recent attacks, such as the Internet worm and its well-known exploitation of system security holes, have caused renewed interest in the development and procurement of highly secure UNIX systems.

USL's approach in addressing the definition of security features has been to work closely with UNIX International (UI) to identify needs and evaluate

functional requirements. A parallel effort has proceeded with government and industry leaders to establish standards through bodies such as IEEE POSIX and X/OPEN. Within these standards bodies USL has been an active participant and has been very successful driving these standards to be consistent with the SVR4.1ES functionality.

Why B2?

SVR4.1ES was designed to meet the rapidly increasing security needs within the UNIX System V community. The TCSEC B2 level seemed a natural fit for System V for several reasons. From a purely functional perspective, the B2 level represents a superset of the features most desirable to government and commercial users. At the B2 level, MAC, ACLs, trusted path, and trusted facility administration are provided. The high level of assurance required to meet the B2 level is sufficient to satisfy the needs expressed by government and commercial customers. The higher levels (B3/A1) provide mostly increased assurance; little is provided in the way of additional function. However, functionality from the B3 and A1 classes was included in the SVR4.1ES feature set. SVR4.1ES TRUSIX-conformant ACLs and trusted facility administration are two examples of B2+ functionality included in SVR4.1ES to meet customer needs. Current and emerging RFPs and standards specifications (ITSEC, EEC security requirements, Navy and Air Force RFPs, and the CISR) all require functions at the B2 level. Most of these requirements also specify ACLs, a B3 feature.

B2 represents what is likely the highest assurance level attainable by a UNIX system (barring a complete system rewrite) while having it still retain the classic UNIX system look and feel and maintain backward compatibility for existing applications.

Due to the modular definition of features and the flexible packaging of SVR4.1ES, function levels from C2 to B2 can be defined. For example, a site may only install the auditing package on the SVR4.1ES base and have C2-level functions or they may install the B1/B2 package and configure the system to be either B1 or B2.

The Architecture of SVR4.1ES

The SVR4.1ES operating system engineering improvements go beyond individual feature development; they involve changes in the structure and architecture of UNIX System V. These changes result in improved maintainability, performance, flexibility, and portability. Typically, though not always, these improvements will be visible only to system porters and not to end users or application developers. Thus, while such improvements may benefit end users and developers, they are of direct interest to UNIX System V source code customers who plan to port or modify the operating system.

The UNIX system has been renowned as a modular, highly portable, operating system. To meet the exacting requirements on operating system modularity at the B2-level, however, the UNIX System V operating system has been further partitioned into modules.

Modularity and Global Variable Reduction

The increased modularity yields great benefits in portability and maintainability. The SVR4.1ES source code tree has been broken down into "common" and hardware-specific trees. Porting code is easier since the "common" code has been identified and separated leaving only the machine-specific code to be ported. Additionally this modular breakdown provides increased maintainability. Since the common and hardware-specific code are now separate, a bug in "common" code needs only to be fixed in a single place. Previously this "common" code may have been shared between several modules, thus the fix would need to be applied several times in several places.

Global variables, variables whose context is spread across several modules, are often in places where a data value needs to be "shared" across several modules, but it is not practical or desirable to break the modules down into functions (and thus pass the data by function call). While easier to implement, global variables can sometimes make maintenance difficult; modification of a variable in one place may have cascading side effects because the variable is used in several other, often undocumented places. With most of the global variables replaced by module-specific variables, modification of a variable in one section of code is far less likely to produce unknown (and unwanted) change in other areas of code where the variables may have been used.

Improved modularity impacts more than the security feature. It improves the entire operating system and benefits all source code customers. Modular code is easier to interpret, maintain, and port. Since future SVR4 platforms such as Enhanced Security / Multiprocessing (ES/MP) are based on the SVR4.1ES source code, the effort required to port from SVR4.1ES to a future release is greatly reduced.

The following sections briefly describe the major innovative security features found in SVR4.1ES: Identification and Authentication Facility (IAF), least privilege/trusted facility administration, MAC, DAC, trusted path, and auditing.

Identification and Authentication Facility (IAF)

This facility provides a framework for modular replacement of identification and authentication schemes. SVR4.1ES provides two different types of schemes: the familiar login/password scheme and a new bilateral scheme called cr1 (Challenge-Response 1). The cr1 scheme implements a simple challenge-response protocol which requires each system to store a cryptographic key for each system with which it will communicate. cr1 supports the Basic Networking Utilities (BNU) and the remote system administration facilities (see subsection *Networking* for a brief discussion of BNU.

Least Privilege/Trusted Facility Administration

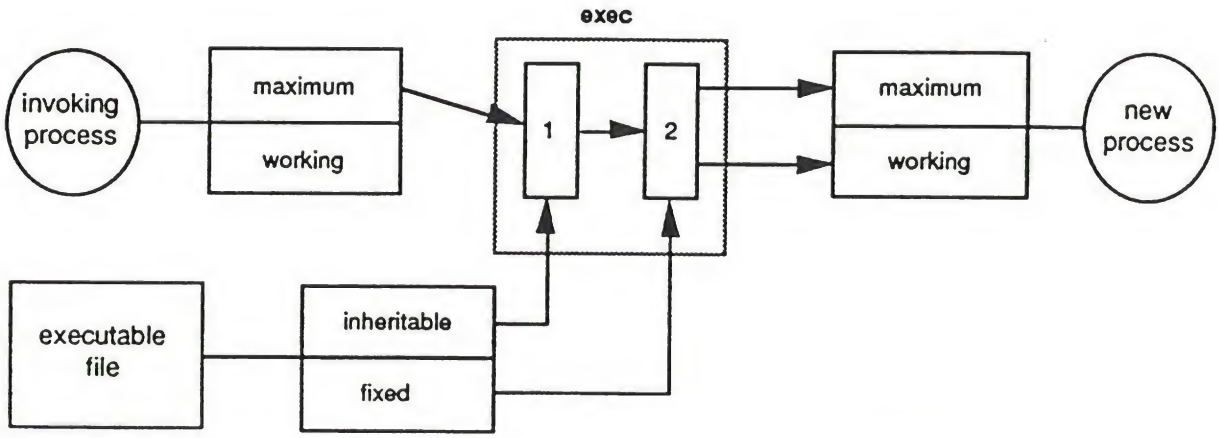
Originally, UNIX systems were small and non-networked. The single-privilege concept worked well because the systems were typically administered and maintained by a single administrator. As UNIX systems became more complex, the tasks of operation, administration, and maintenance were spread among several persons acting in specialized roles. For example, a system operator would be responsible for normal system operations while a system administrator would be responsible for system maintenance. Modifications to the operating system would be done by the systems programmer. In a computer center, environment machine usage charges would be determined by a system auditor. An auditor site security officer might be charged with reviewing the system's audit/accounting trail to determine if any suspicious activity had taken place. Clearly, the tasks executed in these roles are quite different and the level of trust placed in the individuals also varies. For example, the person examining system accounting is clearly not trusted to shut down the system or examine other user's files.

The single-privilege root model does not lend itself well to a role-based scheme.

The alternative is the concept of least privilege, or the ability to execute a task (such as mounting a file system) with the least or minimal amount of privilege required to successfully execute the task. This is clearly beneficial in situations where the individual executing a defined task is not trusted to execute a wide range of tasks. For example, the duties of the operator may be restricted to routine maintenance, such as setting the system date and time. The operator is not permitted to view or modify sensitive system files such as **/etc/shadow**. Following this example, the operator should only be permitted to execute a well-defined set of commands with privilege. Commands outside the defined role of the operator should not be permitted or, if permitted, should execute without any special privilege. Since the UNIX system historically has provided a single privileged identity, that of root (assigned User Id [UID] 0), all tasks requiring use of privilege are executed as root. Thus, the ability to delimit responsibility based on roles is difficult. Workarounds, such as restricted shells, work to a certain extent, but since the restricted environment is itself executing as root, they are always vulnerable and susceptible to attack.

The SVR4.1ES Least Privilege/Trusted Facility Administration feature splits the single privileged role of root into well-defined, less-powerful roles. The SVR4.1ES system provides roles for system operator, security operator, site security officer, and auditor. The feature was designed to be flexible; sites can add additional roles or extend the existing roles as needed. Each role has a well defined set of privileged operations that the user in the role is permitted to execute. For example, the system operator is permitted privileged execution of the **date** command to set the system date and time, but is not permitted privileged execution of the **ed** command to view or modify sensitive system files. What follows is a brief description of how the SVR4.1ES least privilege/trusted facility administration works and how it eliminates some of the more commonly exploitable flaws in the single-privilege mechanism.

In SVR4.1ES, a process has a **maximum** and **working** set of privileges associated with it. The **maximum** set represents the most privilege the process could ever attain, and the **working** set represents the minimum set of privileges required to execute the task. An executable file may have associated with it an **inheritable** or **fixed** set of privileges. A **inheritable** privilege is a privilege that is kept (that is, left "turned on") only if it already existed in the process. A **fixed** privilege is a privilege that is always given to the process, independent of the previous process privileges. When a file is **exec**'ed these sets are computed as illustrated in Figure 1.



(1) intersection of maximum set of privileges of the invoking process with the file's inheritable privileges

(2) union of the results of (1) with the file's fixed privileges

Note: The **fixed** and **inheritable** privilege sets are disjoint; a privilege cannot be present in both sets at the same time.

Figure 1. Computing Privilege Sets

For compatibility with the current UNIX system **setuid** mechanism, SVR4.1ES supports the concept of **fixed** file privileges. When a file that has fixed privilege is executed, those privileges are added (unioned) to the **maximum** privilege set of the invoking process; this forms the **maximum** and **working** privilege sets for the resulting process. Note that the **fixed** privileges are not added to the **maximum** or **working** privilege sets of the invoking process.

For example, if a site determined that all users should be able to execute the ps command and not be subject to mandatory or discretionary access control checks, the administrator would use the **filepriv** command to set the p_DACread and p_MACread privileges as **fixed** privileges. Any user invoking ps would then acquire the p_DACread and p_MACread privileges for the duration of the execution of the ps command.

Trusted Facility Management (TFM)

The trusted facility management (**tfadmin**) facility redefines the way in which the role/privilege assignment mechanism works. In current UNIX systems, an administrator will **login** (or **su**) to an administrative identity. The administrator assumes all file access rights (and privileges in the case of root/UID 0) associated with the identity. All subsequent processes assume these privileges. With this in mind, there are several scenarios by which the vulnerabilities of the system may be exploited. For example, logged in as root, the administrator invokes

$ date 010191 (set system date & time)

$ mail

The administrator assumes that **/bin/date** and **/bin/mail** are being executed. However, since a full pathname was not specified, the administrator is relying on the PATH variable being properly set such that the correct commands are executed.

If the user's path searches **$HOME/my.bin**, and a malicious user was able to plant rogue versions of **date** and **mail** in **$HOME/my.bin**, then the administrator would unknowingly have executed Trojan horse versions of these commands and would have given root access away.

An additional problem is caused by the inheritance mechanism of **exec**. Since all the attributes associated with the root identity are passed to child processes via **exec**, all processes invoked by the administrator execute with privilege, regardless of need. This often results in execution of code not expected to run with root privilege and not designed with trust in mind. This is especially dangerous with commands that in turn execute other commands or that feature escapes to the shell. For example, an administrator escapes to the shell from **mail** and executes **cat**. Since **mail** was running as root, the **cat** command was also executed as root. If a rogue version of **cat** was executed, root privilege has inadvertently been given away.

With **tfadmin** there are no privileges inherent with a single-user identity, rather privileges are associated with a defined role; user identities are then associated with the role. Privilege is acquired by executing **tfadmin**. The **tfadmin** command has an administrative database associated with it. The database contains entries in the following format.

role:alias:command:privilege(s)

For example:

secadmin:date:/bin/date:p_sysops

User identities are then placed in roles; for example, the user Elroy could be placed in the role **secadmin.**

The entry described above allows the user, **Elroy**, in the role of **secadmin**, to execute the command **/bin/date** with the **p_sysops** privilege. For ease of use, the **secadmin** role can use the alias **date** in place of typing in **/bin/date**, which could become tedious.

Based on the previous example, consider the following example:

$ tfadmin date 010191

$ mail

Upon execution, the **tfadmin** command searches its database for an entry for **date** in the role invoking **tfadmin.** If a match is found, the command is executed (via its fully qualified pathname) only with the explicit privileges needed to perform the requested operation. In this case, only the **sysops** privilege is needed to set the date. This is then the only privilege passed to the process executing **date.** The next command, **mail**, requires no privilege to run; therefore, execution via **tfadmin** is unnecessary. Because **tfadmin** will only associate privilege with a defined entry, the command

tfadmin mail

would fail because no database entry would be defined for **mail.**

Mandatory Access Control (MAC)

In order to meet customer needs for high data integrity, Mandatory Access Control labels have been added to SVR4.1ES. With the addition of MAC, all processes, files, and interprocess communication (IPC) objects must have

a security label. The DAC mechanism allows permissions to be set at the discretion of the owner of an object; the owner of a file is able to determine who can (and cannot) access the file. On the other hand, the system administrator sets the MAC mechanism and the system enforces it. The file owner does not set the initial MAC label and is unable to change it.

The SVR4.1ES mandatory access control policy follows a modified Bell-LaPadula model [2] that can be summarized as "read equal or down" and "write equal." Assume a situation where "top secret" dominates "secret" and "secret" dominates "unclassified." Naturally, each level can write to only to files on its own level; but a process at level "top secret" can read files at each class, since it dominates them all. A process at the "secret" level, however, would be able to read files only at "secret" and "unclassified" levels.

Administrators are responsible for determining and setting up the discrete set of labels at which a user can log in as well as setting the login level range on terminal lines. The login level range restricts system access such that when a user attempts to log in, the label the user specified must dominate the login-low label on the terminal line and in turn be dominated by the login-high label on the terminal line. For example, a terminal line with a login level range of "secret" to "top secret" would be unaccessible to a user at "unclassified" (since "unclassified" does not dominate the login-low level "secret").

By default, SVR4.1ES supports 256 classifications and 1,024 categories, though the system can be configured to support up to 65,535 classifications and 2,097,152 categories. For reasons of disk space and performance, SVR4.1ES implements MAC labels with an indirection scheme. Each named classification/category tuple (that is, fully qualified label) is associated with a unique level identifier also known as an LID. The LID serves as a system pointer to the fully qualified label name and is the value stored in the inode. For reasons of user convenience, each fully qualified label may be assigned an alias name. The alias name is a short hand representation of the fully qualified label. For example, the alias for the label

Top Secret:projectA,projectB

may be

TS

The kernel uses the LID as the primary method of label reference. When the kernel is requested to check access, the LIDs involved in the access determination are compared. If write access is requested, the LIDs themselves are simply compared (since the system enforces a policy of write equal and the LIDs are guaranteed to be unique). For example, if write access to a file with a lid of 10045 is requested by a process with an LID of 10045, access is granted since the LIDs are equal. However, if write access is requested to the same file by a process with an LID of 10046, access is denied because the LIDs are not equal. The system supports a policy of read down, so the access check required for a read operation requires an additional step. Since no hierarchy can be determined by the comparing two LIDs (that is, LID 10046 is not guaranteed to dominate LID 10045), the binary representation of the fully qualified labels of the two LIDs needs to be compared. For reasons of system performance, the binary representation of the labels is kept in a cache; its size is a system tunable that may be increased or decreased as required. For example, if a read operation was requested to a file with an LID of 10045 by a process with an LID of 10046, the system would do the following:

- Check to see if the binary representations of the LIDs to be compared were already in the cache.
- If the binary representations of both LIDs were not in the cache, the system would read the LID database and bring the binary representation of the LIDs into the cache.
- The binary representations of the LIDs would be compared to determine if a dominance relationship exists (that is, read access). If so, access would be granted. If not, access would be denied.

An additional form of data integrity, access isolation, has been achieved through the judicious use of mandatory access control levels. By setting up a label hierarchy such that user-defined labels are disjoint (that is, they do not dominate) from system-defined labels, the system is partitioned. Users are prohibited (via MAC) from reading, modifying, or executing sensitive system files, and administrators are protected from inadvertently executing un-trusted code. Figure 2 illustrates how such a lattice may be defined.

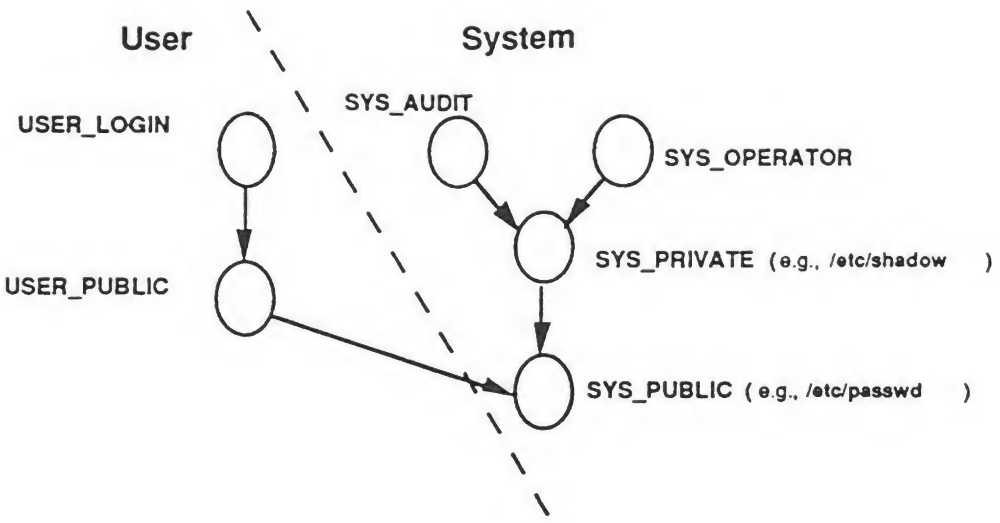


Figure 2. Access Isolation Mechanism

In the lattice depicted above, the levels USER_PUBLIC and USER_LOGIN are defined for non-administrative use. The level USER_PUBLIC is defined for non-administrative user files and commands (emacs, databases, etc.,). The level USER_LOGIN is defined for non-administrative system access; by default, all non-administrative users access the system at this level. The levels SYS_PUBLIC, SYS_PRIVATE, SYS_OPERATOR, and SYS_AUDIT are defined for administrative and system use. The level SYS_PUBLIC is defined for files/commands accessible to administrators and users (such as, **mail, mount, date**). The level SYS_PRIVATE is defined for administrative system access and is not accessible by non-administrative users. The level SYS_AUDIT is reserved for storing the system audit trail.

Considering the lattice defined above, the commands date and mail would be labeled at SYS_PUBLIC. Since both the user and system portions have read access to data labeled at SYS_PUBLIC, both administrators and users have execute permission for these commands. Since the user does not have write permission at the SYS_PUBLIC level (MAC restricts write access), a user cannot plant a Trojan horse at this level. Note that, since the level SYS_PRIVATE dominates SYS_PUBLIC, the administrator does not require either mandatory or discretionary override privilege to access these files. Thus, the administrator executing these commands does not have mandatory access control override permissions and may only execute commands and read files at levels dominated by SYS_PRIVATE. Since the administrator at SYS_PRIVATE does not dominate either USER_PUBLIC

or USER_LOGIN, and does not acquire the privilege required to circumvent mandatory access control, the administrator is protected from invoking Trojan horse programs planted at this level by users.

Discretionary Access Control

SVR4.1ES provides two complimentary DAC mechanisms: UNIX system file permission modes and TRUSIX-conformant Access Control Lists (ACLs). The UNIX file permission modes are retained from previous releases of UNIX System V for compatibility. Users already familiar with UNIX system file permissions will find that this mechanism still works as expected.

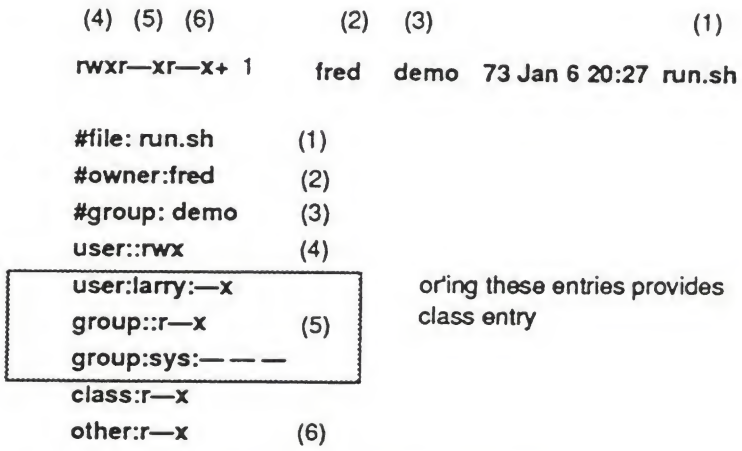
The SVR4.1ES ACLs are designed to satisfy the B3-level Orange Book requirements while still retaining compatibility with the UNIX system file mode scheme. The SVR4.1ES ACL mechanism allows for finer control than do existing file permission bits. It provides the owner of an object the ability to grant or deny access by other users to the granularity of a single user.

For convenience, SVR4.1ES ACLs also allow access rights to members of groups (as defined to the system in the **/etc/group** administrative file). ACLs can also be arbitrarily large; that is, the number of ACL entries is not limited by the system. The system administrator can set the maximum number of entries per ACL by setting a tunable parameter. (Naturally, as ACLs get larger, processing gets slower, which induces a practical limit on the number of ACL entries.)

In SVR4.1ES, an ACL is associated with every file system object and IPC object. ACLs for file system objects are stored in the associated inode; the first seven entries are stored in the inode, and other entries are stored in indirectly-referenced disk blocks. ACLs for IPC objects are stored in an internal structure associated with the instantiation of the IPC object.

An ACL contains all the DAC-access information for the object with which it is associated. For the sake of compatibility, file permissions are displayed as usual in the expected situations, and operations on files behave as they would be expected to on any UNIX System V-based operating system. However, in SVR4.1ES, file permission bits are actually translated into and stored as ACL entries. The ACL entries, which are derived from the file owner, file

owner group, and other permission bits, are called **base** entries. Permission can be granted or denied beyond the **base** entries by including **additional** ACL entries. The concept of the **file group class** ACL entry is critical to understanding SVR4.1ES ACLs. Historically, the middle three permission bits have been used to indicate the permission granted to the file owner's group. In SVR4.1ES, when additional ACL entries are added, these middle three bits, the **file group class**, are used to represent the **maximum** permission granted by an additional ACL entry. This is done for compatibility with existing applications, which depend on **stat** (as opposed to access), to determine access permission. A simple SVR4.1ES ACL would appear as follows. The numbers in parentheses are used to indicate the association between the permission bits, owner and group, and the ACL. They do not appear in SVR4.1ES ACLs.



Notes: + sign indicates file has an associated ACL
The class entry is always equal to the group permission bits. Thus, stating the file provides the maximum permission granted by the ACL

Figure 3. A Simple SVR4.1ES ACL

In the ACL depicted above, the first three entries (prefixed with #) indicate the name of the file (**run.sh**), the owner of the file (**fred**) and the file owner's group (**demo**). The entries for owner, file owner group, and other are derived from the source file's permission bits. The **user** and **group** entries with null ID fields represent the permissions of the owner and owner's group. The entries are NULL so that if the file is given away, the ACL entries (owner and

group) are still correct. The class entry was computed by oring the additional ACL entries for larry, sys and file group class[1].

An ACL consists of the following types of entries, which must be in the following order:

- user entry - This entry is derived from the file owner permission bits; it contains a user ID and the permissions associated with it. There is always one entry of this type. It represents the object owner and is denoted by a null (unspecified) user ID. There may be additional, unique user entries.
- group entry - This entry is derived from the file group permission bits; it contains a group ID and the permissions associated with it. There is always one entry of this type. It represents the object owning group and is denoted by a null (unspecified) group ID. There may be additional, unique group entries.
- other entry - This type of entry contains the permissions granted to a subject if none of the above entries have been matched. There is exactly one of these entries in an ACL.
- class entry - This type of entry contains the maximum permissions granted to the file group class. There is exactly one of these entries in the ACL. The class entry indicates the maximum permission allowed by the ACL. Additionally, this entry acts as a mask and provides compatibility for existing applications. These applications obtain file access permission via stat and attempt to change file status via chmod. For example, see Figure 4

---

[1] The SVR4.1ES ACL mechanism treats the file owner group entry as an additional ACL entry (that is, not a base entry).

Before chmod 000	After chmod 000	After chmod 755 (reset mode bits)
rwxr—xr—x+	————————+	rwxr—xr—x+
#file: run.sh	#file:run.sh	#file: run.sh
#owner: fred	#owner: fred	#owner: fred
#group: demo	#group: demo	#group: demo
user::rwx	user::———	user::rwx
user:larry:——x	user:larry:——x	user:larry:——x
group::r—x	group::r—x	group::r—x
group:sys:——	group:sys:——	group:sys:——
class:r—x	class:———	class:r—x
other:r—x	other:———	other:r—x

Figure 4. Modification of Mode Bits and ACL Using *chmod*

- **default** entry - This type of entry may only exist on a directory. These entries are similar to the entries described above, except that they are never used in an access check. Rather, they are used to indicate the user, **group**, and **other** ACL entries that should be added to a file created within the directory.

Referring to the example above, notice that the ACL entries for file owner, other, and file group class are changed to reflect the intended setting of the permission bits (via **chmod**). No additional ACL entries are modified. The intended effect of the **chmod 000** is accomplished by using the file group class entry as a mask. Note that **chmod** did not modify the file owner group entry. This is because the SVR4.1ES implementation treats the file owner group as an additional ACL entry.

Trusted Path

The SVR4.1ES trusted path feature provides a direct communications path between the user and the system such that all sensitive information entered by the user is in fact being transmitted to the system. Before gaining access to the system, the user must enter a key sequence, the Secure Attention Key (SAK). When the SAK is recognized by the system, a **login** prompt is generated. This differs significantly from historical applications of **login** where initiation only requires the user to enter **return**. The SVR4.1ES trusted path feature provides protection against both system access attacks and Trojan horses.

A common form of system access attack is performed by programs which randomly dial numbers, wait for a tone, and then wait for the system to generate a **login** prompt. These programs then try combinations of **login** and **passwd** to attempt to gain system access. By requiring the user attempting to gain system access to enter a key sequence before generating a **login** prompt,

many simple access programs, keyed on the **login** prompt, are defeated This is because the **login** prompt never appears. More advanced programs have yet another layer of protection (the SAK) to penetrate.

A common form of Trojan horse associated with **login** involves planting a malicious program that masquerades as a **login.** A malicious user will log into the system and start a program which itself poses as **login.** This program then sleeps until an unsuspecting user attempts to gain access to the system. The rogue **login** program then reads the user's login and password, records them, and then **exec's** a version of the real login program. The unsuspecting user has no idea his password has been given away. The SVR4.1ES trusted path facility defeats this type of attack by killing all active processes associated with the **tty** on which the SAK was entered. Thus, in the case of the rogue **login** program, the process posing as **login** is killed, and the user communicates with the true **login** program.

By default, the SAK is a line drop, although the administrator can configure it to be a character or asynchronous line condition, such as a break.

The SVR4.1ES trusted path feature works as follows:

1. A user requesting access to the system enters the SAK.
2. The system identifies the SAK before any line discipline is applied.
3. On detecting the SAK, the trusted computing base (TCB) terminates any current login session, permanently puts open connections in a state such that they can no longer be used for terminal I/O, and eventually reinitiates the login sequence.
4. If login is not completed within the login timeout period, the login program will enter a mode where login interaction cannot proceed until the SAK is entered again.

Auditing

Hand in hand with the ability to penetrate system security is the ability to do so without detection. On most UNIX systems the only record of process execution is the information saved by the UNIX system's process accounting facility. While this data provides some insight into what may have occurred on the system, it can be spoofed and does not provide sufficient data to fully determine an intruder's actions. Additionally, existing UNIX process accounting provides no granularity; it is an all-or-nothing feature. Either

accounting is enabled for all (known) events, for all users, or it is completely disabled. Since recording accounting data is done on an all-event, all-user basis, a lot of system resources are expended. For this reason, it is frequently not used.

These shortcomings have been corrected in SVR4.1ES with the addition of system auditing. Like accounting, auditing records events that occur on the system. However, in addition to simply recording the occurrence of events, auditing also records the parameters associated with the events and the outcome of the events.

Granularity is provided at both the event and user level; that is, the administrator can select specific events which will be audited and can specify the users for whom those events are audited. Since the system's audit daemon runs with a MAC level disjoint from all defined user levels, the presence of the audit daemon (that is, the ability to detect auditing) is undetectable by unprivileged users. SVR4.1ES provides an audit mechanism capable of recording and reporting on all security-related events that occur on the system. All security-related events that occur on the system can be audited, including those events identified as being associated with covert channels.

SVR4.1ES associates most audit events with a system call. For example, the **mk_dir** and **rm_dir** events map to the **mkdir** and **rmdir** system calls. Since system administrators tend to think in terms of system events, SVR4.1ES provides the concept of an event **class**. The class mechanism allows for a logical grouping of event types. For example, the **mk_dir** and **rm_dir** events fall into the **dir_make** class. Since auditing tends to generate large amounts of data, and since an administrator may wish to select most but not all the event types within a class, SVR4.1ES permits selection by both event type and **class**. Additionally the selections can be intermixed (a **class** may be selected and one or more **types** within the class may be turned off).

Since a certain subset of applications may wish to add records to the audit trail, the SVR4.1ES audit feature provides the ability for applications to add their own free-format records to the audit trail. Multiple site or application records may be defined. These added records can be selected and later reported using the standard SVR4.1ES selection and reporting tools.

Events deemed critical to the integrity of the system (that is, events critical to the integrity of the audit trail) are always audited whenever auditing is enabled, regardless of the system-wide and per-user event masks. These events

are called **fixed** events. Other events are auditable at the discretion of the system administrator. They are called **selectable** events.

As stated above, events may be set on either a system-wide or per-user basis. System-wide events are selected by the administrator with the **auditset** command. **auditset** may also be used after auditing is enabled to specify additional events to be audited, or to unselect events that no longer require auditing.

Per-user audit masks may be designated for each user by using the **useradd** command. These masks are permanent—whenever auditing is enabled and the user is logged on, events specified in these masks will be audited. The set of **fixed** events, along with the system-wide and per-user audit masks are **or**'ed to form the user's process audit mask.

Each auditable event, when audited, generates an associated audit record. Collected for each event audited are a time stamp, the user identity, object name, level of the process (subject) causing the event, privileges used, an identification of the type of event, and an indication of the success or failure of the event. Other information specific to the event type is also collected. The **auditrpt** command is used to select, format, and print data from the log file.

Networking

The features described above are all part of the SVR4.1ES Trusted Computing Base (TCB), currently undergoing NCSC evaluation. In its evaluated configuration, SVR4.1ES is a stand-above system; no network capability is present. In many cases, such a configuration is not practical or even realistic. To accommodate environments which require high-security operating system support and interoperability, several modifications have been made to the System V networking utilities for the new security features present in SVR4.1ES. While not part of the evaluated product, these features allow secure network communications within a homogeneous environment.

A number of significant modifications were made in Release 4 which expanded UNIX System V's networking capabilities. These included improvements to existing capabilities that added new features such as the connection server, the identification and authentication facility, and ID mapping. The network enhancements made to SVR4.1ES continue to build

on this foundation by adding the ability to interconnect with both secure and non-secure systems. Note that the ability to interoperate within the same security domain is only possible when interconnecting between SVR4.1ES systems.

Basic Networking Utilities (BNU)

Enhancements were made to BNU for both security and remote administration. Remote administration is a new capability that allows a set of machines to be administered from a single site. Additional enhancements were made to BNU to allow it to take advantage of the security functionality added with SVR4.1ES, specifically MAC. The enhanced BNU allows files to be transferred between two SVR4.1ES systems retaining the same MAC label (for example, Top Secret to Top Secret) or, the MAC label can be mapped (for example,Top Secret can be mapped to Very Secret). As with all the networking utilities, connections can be established to both secure and non-secure systems.

When used with these networking features, the enhanced uucp capabilities enable system administrators to perform remote file transfer and execution tasks while maintaining system security. Additionally, uucp has been redesigned to facilitate use of the connection server and identification and authentication features.

Open Systems Interconnect (OSI)

The full OSI protocol stack has been implemented in the kernel and uses the STREAMS mechanism as a base. Tools are provided for the manipulation of stacks and data. The protocol stack and tools of the implementation conform to the Government Open Systems Interconnect Profile (GOSIP). Also included are tools to assist in the migration from TCP/IP to OSI.

Distributed File Systems

UNIX System V Release 4 (SVR4) introduced support for the two most widely used distributed file systems, the Network File System (NFS) and Remote File Sharing (RFS).

- NFS provides the interoperability and robustness for a network of computers running different operating systems.

- Both RFS and NFS provide the central administration and control of integrated file systems for a network of computers all running UNIX System V.

Additionally, SVR4 provided a set of common administrative utilities to manage both NFS and RFS.

RFS and NFS are protocol-independent and can run over different networks. Diskless operation is supported as well, and the requirement for dedicated root partitions for diskless clients was eliminated in SVR4.

Also in SVR4, NFS could be configured to use secure Remote Procedure Call (RPC) for increasing the security of the network. For SVR4.1ES, more stringent authentication schemes have been implemented for RFS and NFS to provide support for the B2 security features. These enhancements include support for MAC, least privilege, and auditing. NFS and RFS provide support for MAC label mapping. As was the case for BNU, NFS and RFS support the ability to interconnect within a homogeneous and heterogeneous environment. However, the ability to interpret both MAC labels and privilege sets is supported only within a homogeneous environment.

NFS and RFS provide distinct, largely complementary services. Both are implemented within the virtual file system (VFS) framework, and both are administered with a unified set of administrative commands. Generally, a binary program works transparently with any remote or local file system.

RFS allows machines running UNIX System V to selectively share directories containing files, subdirectories, or devices across a network. In Release 4, the RFS feature allowed the server machine to authenticate a client machine when the client attempted to establish a virtual circuit. To strengthen this mechanism, stronger authentication schemes may be specified when a user attempts to establish a virtual circuit with a server.

NFS allows machines to share files and directories selectively across a network.

Functionality on the Horizon

The following is a brief summary of some functionality under consideration for inclusion in future releases of UNIX System V.

Secure X/CMW

One of the most rapidly expanding security markets is the Compartmented Mode Workstation (CMW) market. In simple terms, a CMW system is a "B1" level operating system with a trusted X server and an Information Label. (An Information Label is a MAC-like label assigned to data. It has the unique property of "floating" as data is added to the file.)

Aside from meeting the demands of the CMW and trusted X markets, CMW requirements add a variety of features such as an enhanced trusted path facility, real-time auditing alarms, and a highly secure software distribution facility. Currently, SVR4.1ES is "CMW-ready"; this means that space for the Information Label has been reserved in all internal data structures. By reserving the space beforehand, binary compatibility between the CMW and non-CMW versions of SVR4.1ES is maintained.

Enhanced Cryptographic Support

As the size and speed of networks grows, so does the threat that sensitive data can be accessed by unauthorized users, or that data may be corrupted or compromised. A simple, reliable method of ensuring data security is encryption, the transformation of text into a cipher to conceal its meaning. The non-physical nature of the electronic media makes it necessary to attach some form of encoding to the data in order to properly identify the source, authenticate the contents and provide privacy against electronic surveillance. In small, internally controlled networks, this type of protection is easily provided through a combination of the standard UNIX system crypt facility (to protect the data from disclosure) and simple checksum (to indicate modification of the data). The standard UNIX system tools utilize private key encryption technology. Private Key, as the name implies, requires that the encryption key remain private for the data to be secure. Since the private key is required to both encrypt and decrypt the data the key must obviously be shared between the parties exchanging the data. As networks grow in size and become spread over multiple administrative domains, exchanging the encryption key in a secure manner becomes more and more difficult.

In addition to the problems involved with sharing a private key, the DES algorithm used by System V is on the restricted products list of the U.S. Government, and any product containing DES cannot be exported. Due to these problems, USL has been looking at ways of implementing public key technology in System V. Public Key technology is well suited to large,

public networks since the key used to encrypt the data is public and may be freely shared without compromising the security of the encrypted data.

Kerberos

With the need for network security increasing, the Kerberos authentication system from MIT's Project Athena is being touted as the answer to network security problems. While installation of Kerberos will make a network more secure and represents a significant step toward increasing the overall level of security within a network, it is not by itself the complete solution. With this in mind, USL developed the Network Applications Architecture (NAA). The NAA will use Kerberos technology as an integral component. The version of Kerberos to be included in the USL NAA, as well as the USL defined extensions to Kerberos, are under investigation.

Summary

This paper has presented a brief overview of the computer security marketplace and defined how SVR4.1ES is situated in this market. The market for secure computer systems is here now and will continue to grow well into the 1990's. The market for these systems is governmental and commercial, domestic and international. Comparison to available secure systems indicates that SVR4.1ES is the only system available or in development which provides the features, assurances, and flexibility required to meet the security needs of the commercial and governmental markets worldwide.

Appendix A: Published Security Specifications

Only a few well known and reviewed security specifications are currently in the pubic domain. The predominant security specification is still the TCSEC. In Europe, it is the ITSEC. In the standards arena, TRUSIX, IEEE POSIX, and X/Open have all developed specifications. Following is a brief description of how SVR4.1ES meets the specifications listed above.

TCSEC	SVR4.1ES is currently in formal evaluation at the B2 level. The system meets all the requirements at the B2 level and exceeds the B2 level with its access control lists and trusted facility manager.
ITSEC	The ITSEC provides a mapping of ITSEC requirements to the TCSEC. SVR4.1ES meets the ITSEC F-B2/E4 level.
POSIX (EEC)	SVR4.1ES meets all of the requirements listed as Mandatory and 90 percent of the requirements listed as Optional.

POSIX P1003.6

The IEEE POSIX P1003.6 committee is developing a security interface for portable applications. When complete, P1003.6 will become an addendum to P1003.1. Since P1003.6 must be translated into a language-independent format (it is currently C bindings) before approval as an ISO standard, it is expected that approval will take two to three years. SVR4.1ES will conform to P1003.6. Early analysis indicates that the majority of the functionality required for conformance could be achieved at the command/library level with few operating system modifications required.

Trusted UNIX (TRUSIX) Task Force

In 1987, the NCSC formed the TRUSIX to provide technical assistance to vendors and evaluators involved in developing TCSEC class B3 UNIX systems. The TRUSIX group produced rationale and a worked example in the following areas:

- **auditing** (rationale only, document is currently out for industry review)
- **access control lists**
- **formal model** (review cycle complete final status of document is under determination)

AT&T/USL was the primary contributor to the TRUSIX access control list document, and as such, TRUSIX is entirely consistent with the SVR4.1ES implementation. The formal model was developed specifically for use in UNIX system evaluations and is the model being used in the SVR4.1ES evaluation. The TRUSIX audit document contains rationale only. The rationale contained in the document is entirely consistent with the SVR4.1ES audit implementation.

X/Open

The X/Open Security Working Group(SWG) chose the C2 level of trust as the target for their work. The group focused exclusively in the area of identification and authentication and auditing. Traditionally, the X/Open SWG has had the lead in defining security auditing interfaces. Before the group became dormant, they produced one document (the *X/Open Security Interface Specification: Auditing and Authentication*) which was adopted by POSIX P1003.6 as the base for their auditing specification. The X/Open auditing specification was co-authored by ATT/USL and IBM®. While the interfaces reflect functionality currently present in SVR4.1ES, they also represent functionality requested by various UNIX International special interest groups and by USL's customers.

Appendix B: References

[1] Department of Defense. *Trusted Computer System Evaluation Criteria,* DOD 5200.28-STD, December, 1985.

[2] Bell, D. E. and LaPadula, L. J., *Secure Computer System: Unified Exposition and Multics Interpretation,* MITRE Corporation, MTR-2997, March 1976.

Glossary

ACL	Access Control List
BNU	Basic Networking Utilities
CISR	Commercial International Security Requirements
CMW	Compartmented Mode Workstations
crl	Challenge-Response 1
DAC	Discretionary Access Control
DoD	Department of Defense
EEC	European Economic Community
ES/MP	Enhanced Security/Multiprocessing
GOSIP	Government Open Systems Interconnect Profile
IAF	Identification and Authentication Facility
ISO/JTC	International Standards Organization's Joint Technical Committee
ITSEC	Information Technology Security Evaluation Criteria
JCL	Job Control Language
JSAN	Joint Army and Navy
LID	Label Identification
MAC	Mandatory Access Control
NAA	Network Application Architecture
NCSC	National Computer Security Center
OSI	Open Systems Interconnect

RFP	Request For Proposals
RFS	Remote File Sharing
RPC	Remote Procedure Call
SAK	Secure Attention Key
SWG	X/Open's Security Working Group
TCSEC	Trusted Computer System Evaluation Criteria
TRUSIX	Trusted UNIX Working Group
VFS	Virtual File System

Corporate Headquarters: UNIX System Laboratories, Inc., 190 River Road, Summit, NJ, 07901, USA
Telephone: +1-800-828-UNIX +1-908-522-6000
European Headquarters: UNIX System Laboratories Europe Ltd., International House, Ealing Broadway, London W5 5DB England
Telephone: +11-44-81-587-7711
Asia Pacific Headquarters: UNIX System Laboratories Pacific Ltd., BR Shiba 1 Building, 2-3-18, Shiba, Minato-ku Tokyo 105 Japan
Telephone: +81-3-5484-8601

3-UXT06-1191